University of Patras

Computer Engineering and Informatics Department

Master Program: Hardware and Software of Integrated Systems

Master Thesis

Energy Efficient Instruction Decoding in Application-Specific Instruction-Set Processors

Christos I. Kargas

Supervisor Professor: Constantinos Goutis
Thesis Committee: Constantinos Goutis
Dimitrios Nikolos
Georgios Theodoridis

Patras, 2012

Πανεπιστημιο Πατρων

Τμημα Μηχανικων Η/Υ και Πληροφορικης

Προγραμμα Μεταπτυχιακων Σπουδων

Ολοκληρωμενα Συστηματα Υλικου και Λογισμικου

Μεταπτυχιακη Διπλωματικη Εργασια

Αποκωδικοποίηση Εντολών για Χαμηλή Κατανάλωση Ενέργειας σε Επεξεργαστές Συνόλου Εντολών Ειδικού Σκοπού

Χρήστος Ι. Κάργας

Επιβλέπων Καθηγητής:	Κωνσταντίνος Γκουτης
Τριμελής Επιτροπή:	Κωνσταντίνος Γκουτης
	Δημήτριος Νικολος
	Γεώργιος Θεοδωριδης

Πάτρα, 2012

Energy Efficient Instruction Decoding in Application-Specific Instruction-Set Processors

Christos Kargas

Master Thesis

Computer Engineering and Informatics Department
University of Patras, Greece

and

IMEC / Holst Centre
Eindhoven, The Netherlands

Patras, November 2012

Acknowledgements

The hereby presented master thesis, entitled “Energy Efficient Instruction Decoding in Application-Specific Instruction-Set Processors”, is a description of the research during the Master Thesis and its results, for the Master of Science program “Hardware and Software of Integrated Systems” of the Department of Computer Engineering and Informatics belonging to the Faculty of Engineering of the University of Patras. The study was conducted in cooperation with IMEC/Holst Centre in Eindhoven, The Netherlands.

The assignment and supervision of the present study was conducted by professor Dr. Constantinos Goutis of the Department of Electrical and Computer Engineering of the University of Patras, whom I’d like to thank warmly for his scientific guidance and moral support as well as giving me the great opportunity to perform my thesis abroad.

I would also like to thank deeply principal researcher Jos Huisken for accepting me in IMEC/Holst Centre and his invaluable guidance and counseling during this thesis. Our endless talks on many matters both theoretical and technical have enriched my knowledge, have served me well during my thesis and are sure to be of much value to me later on.

I would also like to express my sincere gratitude to professor Francky Catthoor of IMEC-Leuven for selecting me among the candidates, for the valuable advice and motivation he provided and his inspiring influence. Additionally, senior researcher Jos Hulzink of IMEC/Holst Centre for his assistance in technical matters, sharing his technical knowledge on the development framework used for this thesis and for his accurate and timely suggestions that helped me overcome many major problems in my thesis.

Special thanks to IMEC/Holst Centre for equipment supply and the facilities they provided and Target Compilers for allowing the use of the tools and the feedback on several matters.

I would like to thank senior researcher Mario Konijnenburg of IMEC/Holst Centre for his assistance in several occasions, PhD candidate António Artés Garcia for willfully sharing his knowledge, his technical assistance and always keeping the spirits high along with PhD candidate Bo Liu in the office we shared, Georgios Selimis and Ioannis Giannis for their advice and helping me adapt to the Netherlands and of course all the people in the ULP-DSP group of IMEC/Holst Centre who are always willing to help and for the friendly environment. Finally, Stefania Dakourou for allowing me to use some of her work for comparisons.

Closing, I would like to thank professor Dr. Dimitrios Nikolos of the Department of Computer Engineering and Informatics of the University of Patras and assistant proffessor Dr. George Theodoridis of the Department of Computer and Electrical Engineering of the University of Patras who along with the supervisor Dr. Constantinos Goutis, make up the committeee of my master thesis.

Christos I. Kargas

Patras, November 2012

Ευχαριστίες

Η παρούσα διπλωματική εργασία, με θέμα ¨Αποκωδικοποίηση εντολών για χαμηλή κατανάλωση ενέργειας σε Επεξεργαστές Συνόλου Εντολών Ειδικού Σκοπού (ASIPs) ¨, είναι μία περιγραφή της έρευνας κατά τη διάρκεια της μεταπτυχιακής διπλωματικής εργασίας και των αποτελεσμάτων της, για το Μεταπτυχιακό Πρόγραμμα Σπουδών ‘Ολοκληρωμένα Συστήματα Υλικού και Λογισμικού’ του Τμήματος Μηχανικών Η/Υ και Πληροφορικής της Πολυτεχνικής Σχολής του Πανεπιστημίου Πατρών. Η μελέτη διενεργήθηκε σε συνεργασία με το ερευνητικό κέντρο IMEC/Holst Centre με βάση το Αϊντχόφεν της Ολλανδίας.

Η ανάθεση και η επίβλεψη της παρούσας εργασίας έγινε από τον καθηγητή κ. Κωνσταντίνο Γκούτη, του τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών του Πανεπιστημίου Πατρών, τον οποίο και θα ήθελα να ευχαριστήσω θερμά για την επιστημονική του καθοδήγηση και την στήριξη, καθώς και για την ευκαιρία που μου έδωσε να πραγματοποιήσω την εργασία μου στο εξωτερικό.

Θα ήθελα επίσης θα ευχαριστήσω εκ βαθέων τον επικεφαλή ερευνητή Jos Huiken που με δέχτηκε στο IMEC/Holst Centre καθώς και για την πολύτιμη καθοδήγηση και τις συμβουλές κατά τη διάρκεια της εργασίας. Οι ατελείωτες συζητήσεις μας σε διάφορα θέματα, τόσο θεωρητικά όσο και πρακτικά εμπλούτισαν τις γνώσεις μου, με βοήθησαν πολύ κατά τη διάρκεια της εργασίας και είναι σίγουρο ότι θα μου είναι ακόμα πιο χρήσιμες στο μέλλον.

Θα ήθελα ακόμη να ευχαριστήσω τον καθηγητή Francky Catthoor του IMEC-Leuven που με επέλεξε ανάμεσα στους υποψήφιους, για τις πολύτιμες συμβουλές και την ενθάρρυνση που μου παρείχε καθώς και την έμπνευση που μου ενέπνευσε. Επίσης τον έμπειρο ερευνητή Jos Hulzink του IMEC/Holst Centre για τη συνδρομή του σε τεχνικά θέματα, για τις τεχνικές γνώσεις που μου μετέδωσε στην αναπτυξιακή πλατφόρμα που χρειάστηκε σε αυτή την εργασία καθώς και για τις εύστοχες και έγκαιρες συστάσεις του που με βοήθησαν να ξεπεράσω πολλά μεγάλα εμπόδια στην εργασία μου.

Ένα μεγάλο ευχαριστώ στο ερευνητικό κέντρο IMEC/Holst Centre για τον εξοπλισμό και τις εγκαταστάσεις που μου παρείχε και στην εταιρία Target Compilers που μου επέτρεψε να χρησιμοποιήσω την αναπτυξιακή πλατφόρμα της και για την απάντηση που μου έδωσαν σε διάφορα θέματα όταν επικοινώνησα μαζί τους.

Θα ήθελα ακόμη να ευχαριστήσω τον έμπειρο ερευνητή Mario Konijnenburg του

IMEC/Holst Centre για την βοήθεια του σε διάφορες περιπτώσεις, τον υποψήφιο διδάκτορα Antonio Artes Garcia για την προθυμία του στο να μοιραστεί τις γνώσεις του ή να βοηθήσει σε τεχνικά θέματα μαζί με τον υποψήφιο διδάκτορα Bo Liu στο γραφείο το οποίο μοιραζόμασταν, τον Γεώργιο Σελίμη και τον Ιωάννη Γιάννη για τις συμβουλές τους, τη στήριξη και την βοήθεια που μου προσέφεραν στην προσαρμογή στην Ολλανδία και φυσικά όλους στην ομάδα ULP-DSP του IMEC/Holst Centre για τη προθυμία να βοηθήσουν και για το φιλικό περιβάλλον. Τέλος την Στεφανία Δακουρού, που μου επέτρεψε να χρησιμοποιήσω ένα τμήμα της εργασίας της για διάφορες συγκρίσεις.

Κλείνοντας, θα ήθελα να ευχαριστήσω τον καθηγητή κ. Δημήτριο Νικολό του τμήματος Μηχανικών Η/Υ και Πληροφορικής του Πανεπιστημίου Πατρών και τον καθηγητή κ. Γεώργιο Θεοδωρίδη του τμήματος Ηλεκτρολόγων Μηχανικών και Τεχνολογίας Υπολογιστών του Πανεπιστημίου Πατρών που μαζί με τον επιβλέπων καθηγητή κ. Κωνσταντίνο Γκούτη αποτελούν την εξεταστική επιτροπή της μεταπτυχιακής διπλωματικής εργασίας μου.

Χρήστος Ι. Κάργας

Πάτρα, Νοέμβριος 2012

Abstract

With commercial processor design tools, a designer can quickly design a C-programmable ASIP for a specific application domain. There are several such ASIPs available for both wireless (UWB baseband processing), encryption, and biomedical processing (particularly for ECG beat detection). In traditional CPUs and DSPs the impact of the instruction-set definition and the complexity of the instruction decoder can be substantial, especially in terms of power consumption. Fully orthogonal VLIW processors, do not incur the cost of an instruction decoder that severely. Instead the instruction word becomes very large, thereby shifting the (power-)cost to the program memory or instruction cache. For the purposes of this thesis a SIMD processor is developed and is compared to a soft-SIMD to observe its area, performance and energy efficiency for a bioimaging benchmark and how the processor description in the ASIP language nML, defines the generated HDL. This SIMD processor is turned into orthogonal and using iterative experiments it is investigated, what is the impact on power while manipulating the instruction-set architecture in combination with the program memory size. It is also investigated how instruction-set re-configuration can be exploited to improve power efficiency. Using this investigation guidelines for low-power ASIP design can be produced.

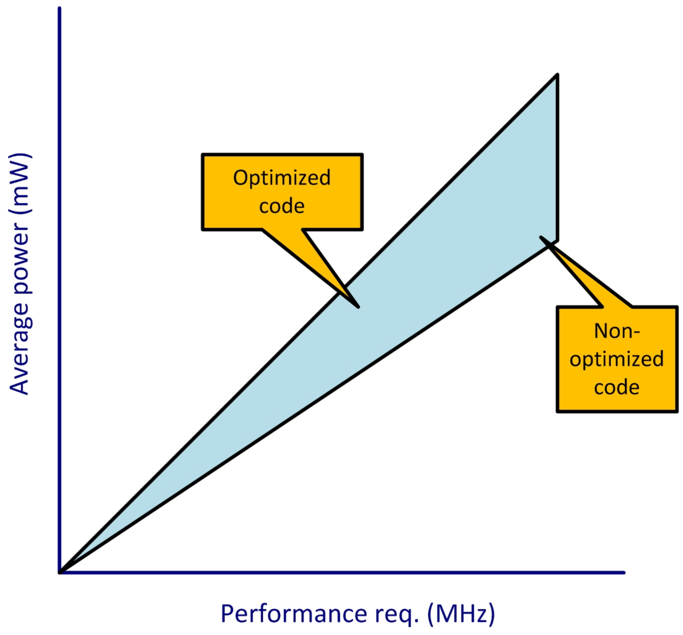
Εκτεταμένη Περίληψη

Με τη σύγχρονη τεχνολογία σχεδιασμού επεξεργαστών, ο σχεδιαστής μπορεί με ευκολία να σχεδιάσει ένα προγραμματιζόμενο Επεξεργαστή Συνόλου Εντολών Ειδικού Σκοπού (ASIP - Application-Specific Instruction-set Processor) για ένα συγκεκριμένο εύρος εφαρμογών. Υπάρχουν διάφοροι τέτοιοι επεξεργαστές διαθέσιμοι για ασύρματες εφαρμογές, κρυπτογράφηση και βιοϊατρικές εφαρμογές (π.χ. στον αλγόριθμο εντοπισμού χτύπου ηλεκτροκαρδιογραφήματος). Στους παραδοσιακούς επεξεργαστές και επεξεργαστές σήματος (DSP - Digital Signal Processor) ο ορισμός του συνόλου εντολών και η πολυπλοκότητα έχουν μεγάλη επίδραση, ειδικά στην κατανάλωση ισχύος. Μία πιθανή λύση σε αυτό το πρόβλημα είναι οι ορθογώνιοι επεξεργαστές μεγάλου μεγέθους λέξης εντολής (VLIW - Very Large Instruction Word).

Με τον όρο *ορθογώνιο επεξεργαστή*, ορίζεται ένας επεξεργαστής οριζόντιου σύνολου εντολών, άρα ένας επεξεργαστής στον οποίο μπορεί να υπάρξει κάθε διαθέσιμος συνδυασμός μεταξύ των διαθέσιμων εντολών και των μεθόδων διευθυνσιοδότησης για πρόσβαση στη μνήμη και το αρχείο καταχωρητών. Οι ορθογώνιοι επεξεργαστές δεν επιβαρύνουν τόσο τον αποκωδικοποιητή εντολών. Αντί αυτού το μέγεθος της λέξης της εντολής γίνεται πολύ μεγάλο, και έτσι μετατίθεται το ενεργειακό κόστος στην μνήμη εντολών προγράμματος (program memory)ή την κρυφή μνήμη εντολών προγράμματος (instruction cache).

Για τους σκοπούς αυτής της διπλωματικής εργασίας, αναπτύχθηκε ένας επεξεργαστής SIMD, ο οποίος συγκρίνεται με έναν soft-SIMD για να μελετηθούν η απαιτούμενη περιοχή στο ενσωματωμένο, επιδόσεις και κατανάλωση ενέργειας για μία βιοϊατρική εφαρμογή, καθώς και το πως η περιγραφή ενός επεξεργαστή στη γλώσσα περιγραφής επεξεργαστών ASIP nML ορίζει την παραγούμενη γλώσσα περιγραφής υλικού (HDL - Hardware Description Language). Ο επεξεργαστής αυτός μετατρέπεται σε ορθογώνιο, και με τη χρήση επαναληπτικών πειραμάτων μελετάται η επίδραση στην κατανάλωση ενέργειας κατά τη διάρκεια αλλαγών στην αρχιτεκτονική του συνόλου εντολών και του μεγέθους της μνήμης εντολών προγράμματος. Ακόμη μελετάται πως μπορεί να εκμεταλλευτεί ο σχεδιαστής την αναδιάρθρωση του συνόλου εντολών για να βελτιώσει την κατανάλωση ενέργειας (*εικόνα 1*).

Οι επεξεργαστές ASIP είναι πολύ διαδεδομένοι τελευταία γιατί μπορούν και συνδυάζουν επιδόσεις, χαμηλή κατανάλωση ενέργειας και κυρίως ευελιξία. Με αυτούς μπορούμε πιο εύκολα να βρούμε τη χρυσή τομή ανάμεσα στα διάφορα

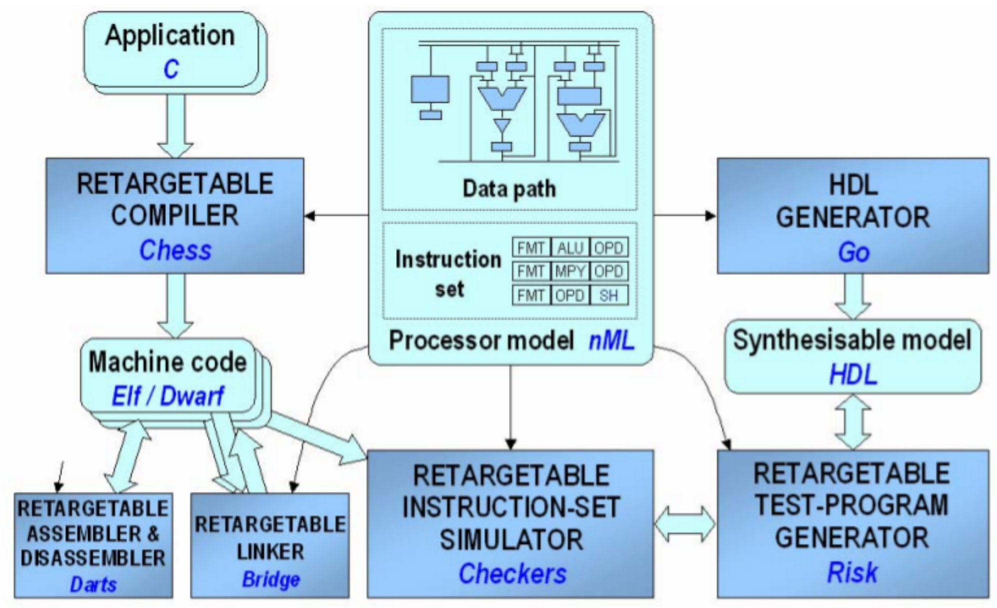


Εικόνα 1: Μέση κατανάλωση ισχύος και συχνότητα λειτουργίας επεξεργαστών ανάλογα με το επίπεδο βελτιστοποίησης του κώδικα σε εμπορικούς επεξεργαστές ASIP

χαρακτηριστικά του επεξεργαστή που θέλουμε, με αποτέλεσμα έναν επεξεργαστή αρκετά ισχυρό και ενεργειακά αποδοτικό για το εύρος εφαρμογών που θέλουμε. Δεν πρέπει να ξεχνάμε ότι στη σύγχρονη αγορά ενσωματωμένων συστημάτων, η κατανάλωση ενέργειας είναι ένα από τα μεγαλύτερα προβλήματα. Ωστόσο, καθώς η τεχνολογία προχωράει με αλματώδεις ρυθμούς, μερικές από τις λύσεις που έχουν δώσει οι σχεδιαστές σε διάφορα προβλήματα που αντιμετώπιζαν στο παρελθόν μελετούνται ξανά, με σκοπό να βρεθεί μία πιο αποτελεσματική λύση.

Για την ανάπτυξη τους, οι επεξεργαστές ASIP χρησιμοποιούν ένα ειδικό περιβάλλον ανάπτυξης με εξελιγμένα εργαλεία ανάπτυξης (*εικόνα 2*), με τα οποία η ανάπτυξη ενός επεξεργαστή ASIP μπορεί να γίνει σε πολύ λιγότερο χρόνου από όσο χρειάζεται ένας κανονικός επεξεργαστής. Προσφέρουν μία πλήρη σουίτα για το σχεδιασμό του επεξεργαστή από το σχεδιασμό της αρχιτεκτονικής του επεξεργαστή, τον ορισμό του συνόλου εντολών και των λειτουργιών μέχρι την προσομοίωση εφαρμογών και την παραγωγή γλώσσας περιγραφής υλικού. Οι διαθέσιμες δυνατότητες προσομοίωσης μπορούν να συνδυαστούν με τις δυνατότητες αναγνώρισης χαρακτηριστικών (profiling)για να ερευνηθούν διαφορετικές υλοποιήσεις και μέσω επαναληπτικών αλλαγών στον αρχικό κώδικα να βρεθεί η ιδανική υλοποίηση για ένα συγκεκριμένο εύρος εφαρμογών.

Πολλές φορές η κατανάλωση ισχύος και η κατανάλωση ενέργειας συγχέονται και θεωρείται ότι είναι το ίδιο. Αυτό είναι λάθος. Η χαμηλή κατανάλωση ισχύος δεν μας εξασφαλίζει και χαμηλή κατανάλωση ενέργειας. Στα σύγχρονα ενσωματωμένα συστήματα, η ενέργεια είναι ο καθοριστικός παράγοντας που πολλές φορές καθορίζει την επιτυχία ενός επεξεργαστή. Η χαμηλή κατανάλωση ισχύος και η υψηλές



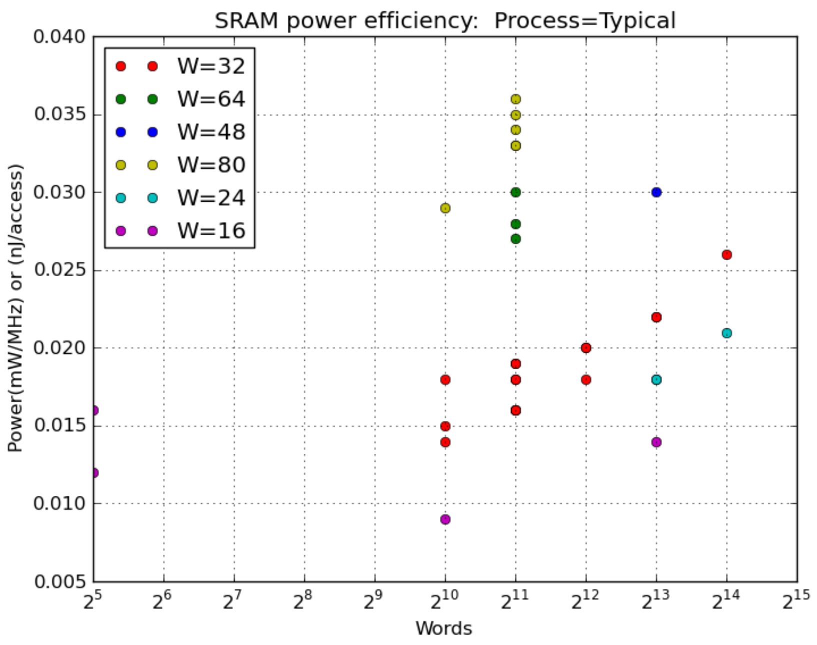
Εικόνα 2: Επισκόπηση των εργαλείων ανάπτυξης επεξεργαστών ASIP της Target που χρησιμοποιήθηκαν για αυτή την εργασία

επιδόσεις από μόνες τους δεν είναι αρκετές, καθώς αυτά τα δύο πολλές φορές έρχονται σε αντίθεση. Ένας επεξεργαστής που σχεδιάζεται για χαμηλή κατανάλωση συχνά δεν έχει καλές επιδόσεις ή αναγκάζεται να λειτουργεί σε χαμηλή συχνότητα, με αποτέλεσμα να χρειάζεται περισσότερη ενέργεια για να ολοκληρωθεί μία εργασία. Άρα, ένας σχεδιαστής επεξεργαστών πρέπει να κάνει ένα συμβιβασμό και να κρατήσει μία ισορροπία ανάμεσα στην ισχύ και την επίδοση, ανάλογα με την εφαρμογή για την οποία προορίζεται. Ευτυχώς, στους επεξεργαστές ASIP , είναι εύκολο και γρήγορο να γίνουν αλλαγές που αλλάζουν αυτές τις ισορροπίες.

Υπάρχουν διάφορες τεχνικές για να μειωθεί η κατανάλωση ισχύος, και οι οποίες μπορούν να εφαρμοστούν σε διαφορετικά επίπεδα της σχεδίασης (επίπεδο μεταγλωττιστή, αρχιτεκτονικής ή κυκλωμάτων), όπως δυναμική κλιμάκωση τάσης DVS - Dynamic Voltage Scaling, clock gating , ή κωδικοποίηση δεδομένων. Κάθε μία από αυτές έχει τα δικά τις πλεονεκτήματα και είναι στη διάθεση του σχεδιαστή να επιλέξει ποια αυτές να χρησιμοποιήσει.

Συνήθως θεωρείται δεδομένο ότι όσο πιο μικρή η μνήμη που χρησιμοποιεί ένας επεξεργαστής, τόσο καλύτερα. Αλλά αυτό που παραβλέπεται είναι η επιβάρυνση που έχει η χρήση της μικρής μνήμης στα υπόλοιπα μέρη του σχεδίου. Για ακριβώς αυτό το λόγο, διερευνάται σε αυτή την εργασία το κατά πόσο μία μεγαλύτερου μήκους λέξη εντολής μπορεί να αποδώσει καλύτερα για το σύνολο του συστήματος, παρά το επιπλέον κόστος στη μνήμη, καθώς αυτή θα επέτρεπε στον σχεδιαστή να χρησιμοποιήσει τα επιπλέον διαθέσιμα ψηφία της λέξης εντολής για να προσαρμόσει το σύνολο εντολών ακριβώς στην εφαρμογή για την οποία προορίζεται ο επεξεργαστής και κατόπιν να επανακωδικοποιηθεί με πιο αποτελεσματικό τρόπο.

Μελέτες σε διάφορες σύγχρονες εμπορικές SRAM μνήμες δείχνουν ότι παρότι οι μεγαλύτεροι πίνακες μνημών (που αποτελούνται από άλλους μικρότερους υποπίνακες) τείνουν να προσφέρουν περισσότερο αποθηκευτικό χώρο για την περιοχή που διεκδικούν στο ολοκληρωμένο, η κατανάλωση ισχύος (αναφορικά με την συχνότητα λειτουργίας) τείνει να αυξάνεται με ταχύτερους ρυθμούς όσο το μέγεθος της λέξης αποθήκευσης ή ο συνολικός χώρος αποθήκευσης αυξάνεται (*εικόνα 3*). Αυτό συνεπάγεται ότι ενώ οι μικρότερες μνήμες είναι καλύτερες σε κατανάλωση ισχύος, οι μεγαλύτερες μνήμες μπορούν να προσφέρουν πολύ περισσότερο αποθηκευτικό χώρο για αναλογικά λιγότερη απαιτούμενο χώρο στο ολοκληρωμένο, το οποίο με τη σειρά του έχει άμεση επίδραση στην κατανάλωση ενέργειας. Η καλύτερη λύση είναι μία ενδιάμεση τιμή μεγέθους λέξης και μεγέθους μνήμης που εγγυάται υπεραρκετό αποθηκευτικό χώρο (ειδικά για τη μνήμη των εντολών εκτέλεσης) με αναλογικά μικρότερο ενεργειακό κόστος.

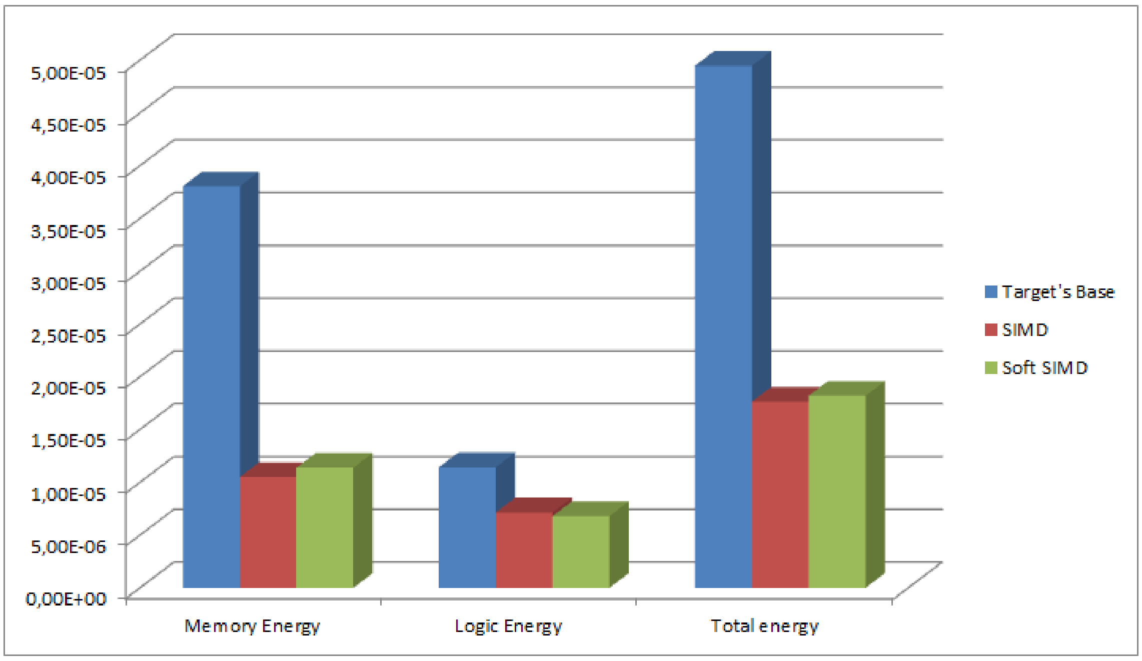


Εικόνα 3: Μέση κατανάλωση ενέργειας με διαφορετικό μέγεθος λέξης και μνήμης για εμπορικές SRAM

Η αρχιτεκτονική του συνόλου εντολών είναι ένα από τα κυριότερα και πιο καθοριστικά χαρακτηριστικά ενός επεξεργαστή. Αυτό ισχύει και για τους επεξεργαστές ASIP, όπου τα εργαλεία λογισμικού του παράλληλου σχεδιασμού υλικού και λογισμικού (hardware/software co-design) πρέπει να προσαρμοστούν σε αυτό το σύνολο εντολών και να το υποστηρίξουν.

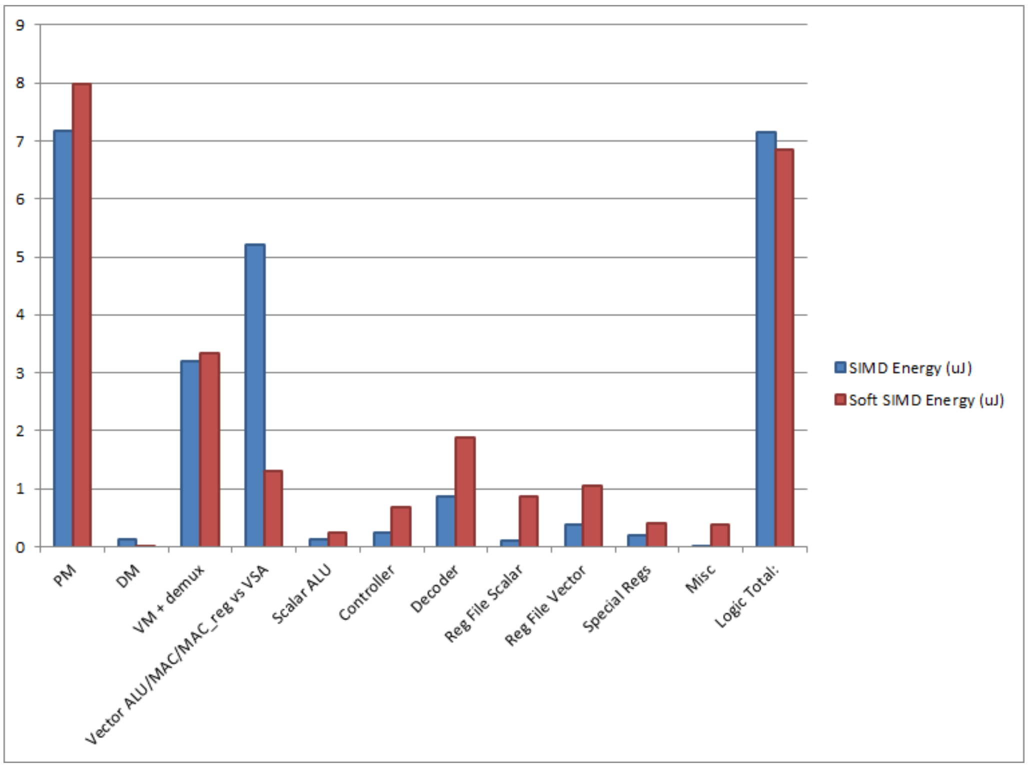
Μία από τις πιο σημαντικές αρχιτεκτονικές επεξεργαστών είναι οι αρχιτεκτονικές SIMD (Single Instruction Multiple Data). Αυτές επιτρέπουν την καλύτερη αξιοποίηση παράλληλων εντολών εκτέλεσης (instruction-level parallelism). Υπάρχουν διάφορες προσεγγίσεις για την υλοποίηση μίας τέτοιας αρχιτεκτονικής, με κυριότερες την υλοποίηση της σε υλικό (hardware SIMD) ή λογισμικό (software SIMD). Η αρχιτεκτονική hard-SIMD στηρίζεται σε πολλαπλές μονάδες εκτέλεσης στο υλικό ενώ η soft-SIMD μεταφέρει αυτήν την πολυπλοκότητα στον μεταγλωττιστή του κώδικα εκτέλεσης και τον αποκωδικοποιητή του συνόλου εντολών.

Κατά τη διάρκεια αυτής της διπλωματικής αναπτύχθηκε ένας επεξεργαστής ASIP με αρχιτεκτονική hard-SIMD ο οποίος και συγκρίθηκε με έναν απλό επεξεργαστή και έναν soft-SIMD με σκοπό να διερευνηθούν οι επιδόσεις σε ενέργεια και απαιτούμενο χώρο σε ολοκληρωμένο. Σαν εφαρμογή ελέγχου της επίδοσης χρησιμοποιείται μία βιοϊατρική εφαρμογή βασισμένη σε ένα Γκαουσιανό φίλτρο. Τα αποτελέσματα δείχνουν ότι και οι δύο υλοποιήσεις επεξεργαστών SIMD έχουν σαφώς καλύτερες επιδόσεις από τον απλό επεξεργαστή (*εικόνα 4*). Εφόσον μπορούν να εκτελέσουν την εφαρμογή πιο γρήγορα με μικρότερο αναλογία κατανάλωσης ισχύος και συχνότητας λειτουργίας, έχουν και σαφώς μικρότερο ενεργειακό κόστος.



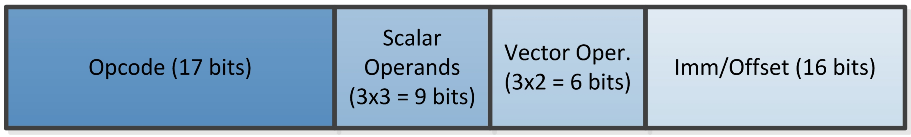
Εικόνα 4 Συγκρίσεις για τις καταναλώσεις ενέργειας για τις μνήμες, τα μέρη λογικής και συνολικά για τις τρεις διαφορετικές αρχιτεκτονικές

Η σύγκριση μεταξύ των δύο υλοποιήσεων hard-SIMD και soft-SIMD είναι κάπως δύσκολη, καθώς διαφέρουν σε πολλά σημεία, αλλά επικεντρώνεται στην κατανάλωση ενέργειας γιατί αυτό είναι το σημείο που μας ενδιαφέρει περισσότερο. Η αρχιτεκτονική hard-SIMD απαιτεί λίγο μικρότερο ποσό ενέργειας για την βιοϊατρική εφαρμογή, με μικρότερη περιοχή απαιτούμενων κελιών στο ολοκληρωμένο. Υλοποιεί στο υλικό της τέσσερις πολλαπλασιαστές των 16 ψηφίων για τον υπολογισμό των λειτουργιών που χρειάζεται η μονάδα MAC (Multiply-Accumulate), οι οποίοι είναι πολύ απαιτητικοί τόσο σε χώρο στο ολοκληρωμένο όσο και σε ισχύ. Παρόλα αυτά έχει ένα συνολικά σχετικά απλό σχεδιασμό. Η αρχιτεκτονική soft-SIMD έχει αρκετές πολύπλοκες βελτιστοποιήσεις με κυριότερες ένα εξελιγμένο αρχείο καταχωρητών για αποθήκευση διανυσματικών δεδομένων (Vector Vegister File), μία μονάδα επεξεργασίας διανυσματικών δεδομένων μέσω επαναληπτικών ολισθήσεων και προσθέσεων (vector shift-add unit) που υποκαθιστά τους πολλαπλασιαστές MAC και λέξη εντολής μήκους 80 ψηφίων που μπορεί να προσαρμοστεί σε διαφορετικά μεγέθη υπολέξεων. Μελετώντας τα αποτελέσματα μετρήσεων ενέργειας για τα διάφορα μέρη της κάθε αρχιτεκτονικής (*εικόνα 5*), είναι εμφανές ότι η πολυπλοκότητα που επιβάλλουν οι προσθήκες της αρχιτεκτονικής soft-SIMD στα υπόλοιπα μέρη του επεξεργαστή και κυρίως στον αποκωδικοποιητή εντολών (ο οποίος πρέπει να αποκωδικοποιήσει μία πολύ μεγάλη λέξη εντολής των 80 ψηφίων) τα επιβαρύνει πάρα πολύ με αποτέλεσμα να αυξάνεται δραματικά η απαίτηση σε χώρο και κατανάλωση ισχύος. Από αυτά συμπεραίνουμε ότι καθαρά από την οπτική πλευρά της κατανάλωσης ενέργειας, η αρχιτεκτονική hard-SIMD με τον πιο απλό και ξεκάθαρο σχεδιασμό της που εξυπηρετεί στην γρήγορη ανάπτυξη και αποσφαλμάτωση του επεξεργαστή και παρέχει πολύτιμο χώρο για βελτιώσεις στην σχεδίαση και κυρίως κωδικοποίηση του συνόλου εντολών μέσω πειραματικών βελτιστοποιήσεων, κάτι που αξιοποιεί πλήρως τις δυνατότητες των εργαλείων ανάπτυξης επεξεργαστών ASIP.



Εικόνα 5: Συγκρίσεις κατανάλωσης ενέργειας μεταξύ των διαφορετικών τμημάτων των αρχιτεκτονικών hard-SIMD και soft-SIMD

Η αρχιτεκτονική hard-SIMD κατόπιν μετατράπηκε σε αρχιτεκτονική με ορθογώνιο σύνολο εντολών, και το μήκος της λέξης εντολής από 16 ψηφία έγινε 48. Η αρχική ιδέα ήταν να γίνει η αρχιτεκτονική του συνόλου εντολών όσο γίνεται μεγαλύτερη και πλήρως ορθογώνια, δίνοντας έτσι τη δυνατότητα πλήρους ελέγχου στα παραγόμενα σήματα αλλά αυτό δεν κατέστη δυνατόν λόγω τον περιορισμένων δυνατοτήτων των τρεχόντων εκδόσεων των εργαλείων ανάπτυξης. Η νέα λέξη εντολών μήκους 48 ψηφίων (εικόνα 6) περιέχει 17 ψηφία για την επιλογή της εντολής και των παραμέτρων λειτουργίας της (opcode), 3*3=9 ψηφία για τους τρεις απλούς τελεστές, 3*2=6 ψηφία για τους διανυσματικούς τελεστές και 16 ψηφία για άμεση εισαγωγή τιμών(offset).

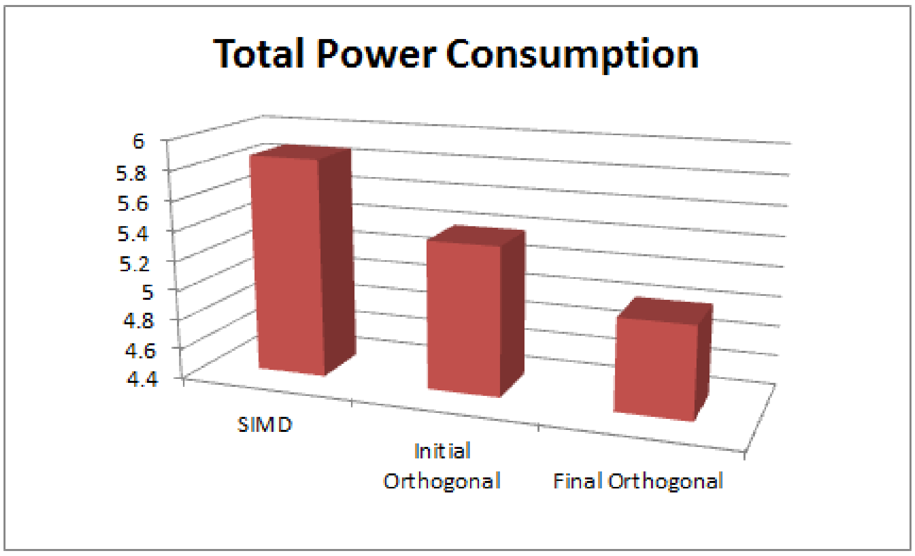


Η ορθογώνια αρχιτεκτονική συνόλου εντολών μήκους 48 ψηφίων

Η νέα ορθογώνια αρχιτεκτονική, παρά το τέσσερις φορές μεγαλύτερο μήκος λέξης εντολής και άρα και μέγεθος μνήμης εντολών προγράμματος είναι 10% πιο αποδοτικό ενεργειακά σε σχέση με την προηγούμενη αρχιτεκτονική (soft-SIMD), καθώς μειώνεται κατά πολύ η κατανάλωση ενέργειας του αποκωδικοποιητή εντολών, της μονάδας MAC και των άλλων τμημάτων.

Διεξήχθησαν διάφοροι πειραματισμοί για το πως θα μπορούσε να μειωθεί η πολυπλοκότητα του αποκωδικοποιητή μέσω της κωδικοποίησης του συνόλου εντολών. Αυτό πετυχαίνεται με τη χρήση των αρκετών ψηφίων στο μήκους 17 ψηφίων opcode που δεν χρησιμοποιούνται, με σκοπό να βελτιωθεί η γραμματική nML που περιγράφει το σύνολο εντολών και μέσω αυτής ο παραγόμενος κώδικας περιγραφής υλικού και να γίνει πειραματισμός με διαφορετικές μεθόδους κωδικοποίησης και μήκη λέξης εντολής. Τα εργαλεία ανάπτυξης ASIP παρέχουν επιλογές για παρακολούθηση της εκτέλεσης ενός προγράμματος και εξαγωγή στατιστικών στοιχείων, κυρίως για τα ποσοστά εκτέλεσης της κάθε εντολής. Αυτά τα στοιχεία, σε συνδυασμό με τη δυνατότητα γρήγορης παραγωγής νέου κώδικα περιγραφής υλικού που δίνουν τα εργαλεία ανάπτυξης ASIP, επιτρέπουν την εξερεύνηση διαφορετικών επιλογών χωρικής ή χρονικής τοπικότητας και των επιπτώσεων τους στην αρχιτεκτονική και την κατανάλωση ενέργειας που αυτή απαιτεί. Το μόνο πρόβλημα είναι ότι χρειάζεται ένας αυτοματοποιημένος τρόπος για να μπορεί γρήγορα μία νέα περιγραφή nML να περάσει από τα διάφορα στάδια μεταγλωττίσεων, προσομοιώσεων και επαληθεύσεων μέχρι την παραγωγή κώδικα περιγραφής υλικούς και καθώς την σύνθεση του με στοιχεία βιβλιοθηκών ολοκληρωμένων κυκλωμάτων και την εξαγωγή στοιχείων για την κατανάλωση ενέργειας και την μετατροπή αυτών σε φιλική και ευανάγνωστη για τον χρήστη μορφή. Αυτό λύνεται με τη χρήση διαφόρων σεναρίων εκτέλεσης (scripts) ή προγραμμάτων από διάφορες γλώσσες προγραμματισμού.

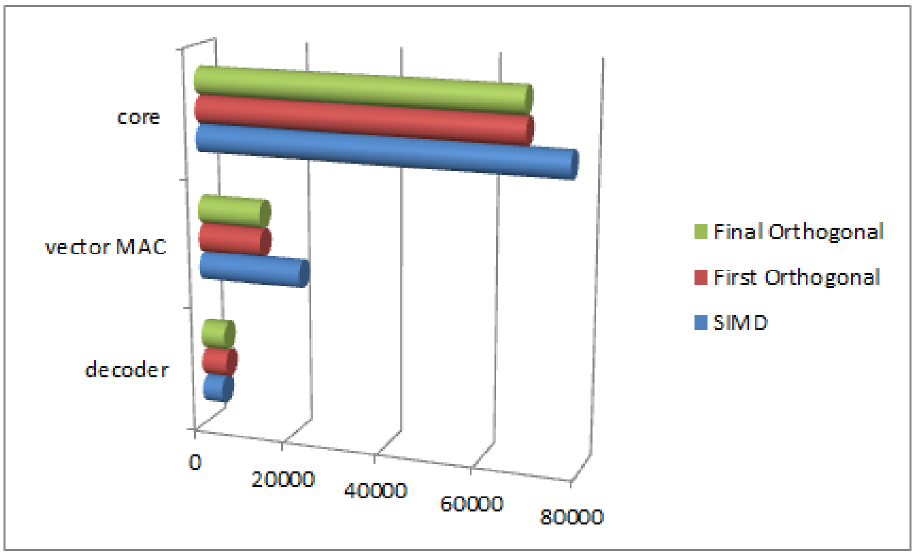
Η τελική υλοποίηση σαν αποτέλεσμα των προαναφερθέντων μεθόδων, διατηρεί τον βασικό περιορισμό ότι δεν πρέπει να γίνουν αλλαγές στην βασική λειτουργικότητα και το πλήθος των εντολών, γιατί παρόλο που αυτό θα έδινε πολλές παραπάνω δυνατότητες, ανοίγει τον 'ασκό του Αιόλου' για την πολυπλοκότητα της αρχιτεκτονικής και των πειραματισμών. Μέσω της χρήσης διαφορετικής κωδικοποίησης για τις εντολές που ανήκουν σε βρόγχους επαναλήψεων που εκτελούνται πάνω από το 95% στο σύνολο των εντολών, και των βελτιστοποιήσεων της κωδικοποίησης τους τόσο συνολικά όσο και μεταξύ τους, προκύπτει ότι υπάρχει τεράστια μείωση στην δραστηριότητα αλλαγής ψηφίων (toggling activity) του αποκωδικοποιητή εντολών των μνημών, τα οποία με τη σειρά τους βελτιώνουν την συνολική κατανάλωση ενέργειας κατά 8% σε σύγκριση με την αρχική ορθογώνια αρχιτεκτονική και 15% σε σύγκριση με την αρχιτεκτονική SIMD *(εικόνα 7)*.



Εικόνα 7: Συνολική κατανάλωση ενέργειας την αρχιτεκτονική SIMD, την αρχική και την βελτιστοποιημένη αρχιτεκτονική ορθογώνιου συνόλου εντολών

Είναι προφανές ότι η κωδικοποίηση του συνόλου εντολών και του μήκους λέξης εντολής αποτελεί ένα σημαντικό παράγοντα στην ενέργεια και τις επιδόσεις ενός επεξεργαστή. Με την χρήση των εργαλείων σχεδίασης ASIP και των δυνατοτήτων γρήγορου επανασχεδιαμού που μας παρέχουν μπορούμε να προσαρμόσουμε το σύνολο εντολών στην εφαρμογή για την οποία προορίζεται, με αποτέλεσμα να μειώνεται δραστικά η κατανάλωση ενέργειας χωρίς απώλεια στις επιδόσεις. Το μήκος της λέξης εντολών ορίζει τη σχέση μεταξύ της πολυπλοκότητας των μνημών και της συνολικής αρχιτεκτονικής (μέσω του αποκωδικοποιητή εντολών) *(εικόνα 8)*. Μία μικρότερου μήκους λέξη εντολής χρειάζεται μικρότερη μνήμη και μεταφέρει την πολυπλοκότητα στον αποκωδικοποιητή εντολών, ενώ μία μεγαλύτερη λέξη εντολής προϋποθέτει έναν απλούστερο αποκωδικοποιητή αλλά μεγάλες μνήμες. Όμως, μερικές φορές και ανάλογα το πεδίο εφαρμογών η μείωση της δραστηριότητας toggling των εντολών είναι πιο σημαντική από από το μήκος τους.

Συνδυάζοντας τα δεδομένα από τους πειραματισμούς στις διαφορετικές αρχιτεκτονικές ASIP με τα δεδομένα από τις μετρήσεις στις μνήμες, προκύπτει το συμπέρασμα ότι καθώς η ενέργεια είναι καθοριστικός παράγοντας στη σχεδίαση, ένας επεξεργαστής και κυρίως ένας επεξεργαστής ASIP μπορεί να ωφεληθεί πάρα



Εικόνα 8: Απαιτούμενος χώρος σε ολοκληρωμένο (σε χώρο κελιών) για την αρχιτεκτονική SIMD και τις δύο αρχιτεκτονικές με ορθογώνιο σύνολο εντολών

πολύ από ένα μεγαλύτερο σε μήκος λέξης και όχι υπερβολικά συμπιεσμένο σύνολο εντολών, το οποίο μπορεί να μειώσει την συνολική πολυπλοκότητα σε όλη τα επιμέρους τμήματα της αρχιτεκτονικής, άρα και την απαιτούμενη ενέργεια χώρο σε ολοκληρωμένο, και παρέχει τη δυνατότητα βελτιστοποιήσεων για συγκεκριμένες εφαρμογές χωρίς να επιβαρύνει πολύ τις μνήμες. Πάραυτα, η αύξηση του μήκους της λέξης εντολών πρέπει να γίνει ελεγχόμενα λόγω της εκθετικά αυξανόμενης κατανάλωσης ισχύος πάνω από το μήκος λέξης 64 ψηφίων. Είναι στην ευχέρεια του σχεδιαστή να επιλέξει το κατάλληλο μέγεθος λέξης εντολής για την κάθε περίπτωση.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Embedded systems have developed a lot the last decade, with a main research focus on improving performance and cost and secondary on improving power, flexibility or reliability. However, the same principles do not apply on the market anymore. Nowadays, one of the features that has come to pose a big problem is that of power and energy consumption. By rapidly increasing the performance, we have increased the energy consumption of the system at an even greater rate. Application Specific Instruction-set Processors were developed and introduced to SoC design for their unique feature to combine performance, flexibility and power efficiency. There has been a lot of work in the field of performance, but in the field of power there is still a lot of space for improvement.

The motivational framework for this study is summarized in the following sections, including an introduction to embedded systems, processors and especially ASIPs that provides the reader with essential information for the next chapters

1.1.1 Embedded systems

Embedded computer systems are computers inside devices designed to perform certain functions and are one of the most rapidly growing part of the computer industry. These devices are everyday machines excluding laptops, desktops or servers (which are by comparison general purpose) and they range from mobile phones, hand-held digital devices or video consoles to cars and washing machines. Their purpose is to perform a specific task as efficiently as possible (perceived by the common user as “fast” or “responsive”) while providing a certain autonomy on battery or generally low power consumption. They are usually programmed to perform just a handful of functions or programmable but within small limits. The parts of the code that are used most often are heavily optimized. More complicated systems come with firmware which can be updated for bug corrections or added functionality. That is how they manage to uphold time-to-market deadlines and reduce the cost.

Compared to desktops, embedded systems differ in a lot of ways. They have a much broader range of technical characteristics since each one is developed for a precise function, from low-end and cheap 8-bit or 16-bit embedded microprocessors to way high-end, efficient and expensive embedded microprocessors that provide greater functionality and maximum performance. Also, unlike desktops, each embedded system is developed with predefined criteria, usually in performance, area, responsiveness, or energy consumption, with more narrow limits.

Embedded systems are also used for *real-time systems* where they are programmed to meet specific real-time constraints, meaning that the time from certain events until the proper response of the system cannot exceed a set time limit. Those real-time systems can be either soft or hard. In hard real-time systems (e.g. in planes), failing to keep up to the deadline results in a complete system failure, so the design and standards of those systems are far more strict, or soft real-time systems where a few failures are tolerated but they reduce the quality of service.

Typically, the purpose of an embedded system is to process information in the form of signals, so one of the most common embedded systems are *digital signal processors* (DSPs). The term “signal” does not necessarily denote a telecommunication transmission, but it could also be a video, an image, a sound or any form of data. DSPs are specialized processors optimized for for digital signal processing algorithms. These algorithms are from many domains, from transforms (e.g. Discrete Cosine Transform, Fast Fourier Transform), time-domain filtering (e.g. finite impulse response or infinite impulse response filters) or convolution to error correction. But in all those the core unit is common: the multiply-accumulate operation. A characteristic example is the Finite Impulse Response (FIR) filter:

$$Y[n] = \sum X[n-k]h[h]$$

where $X[n]$ is the sampled input, $h[k]$ are the filter coefficients that characterize the particular filter and $Y[n]$ is the output.

As indicated above, the filter is composed of registers, multipliers and an adder, therefore the core that is repeated is comprised of subsequent additions of a product. For the DSP to be effective, this core has a dedicated hardware unit to perform the multiply-accumulate operation (MAC). So a MAC instruction of “MAC A,B,C” actually implies “A = A + B * C”. There has been a lot of research in trying to optimize the MAC unit, because in many cases that is where the bottleneck of the whole system lies. Another common way used to accelerate communication algorithms is by optimizing encoding and decoding forward error correction codes.

Performance versus power consumption

In contrast with the desktop/server market, in the embedded market power consumption and production cost play a much greater role. Desktop and server systems have a stable power supply whereas most embedded systems rely on battery supply. Therefore, embedded systems compared to desktop ones are not only constrained in terms of cost, physical area and memory size but also in energy consumption. As a result the designers have to measure carefully the metrics of their system, weighting performance against energy consumption. For example, it would be inefficient to produce an embedded system that has great performance, but drains the battery really fast, or the other way around, one that has an excellent low power consumption but takes a really long time to perform its given task.

To measure effectively and accurately the performance and power consumption, specific benchmarks are used (like EEMBC). Figures 1.1 and 1.2 show the relative performance per watt of typical operating power and raw performance compared a specific processor respectively. From these two figures it is very interesting to notice certain points that stand out. For example, the NEC VR 4122 is probably the best one in terms of performance per watt, especially for the telecomm benchmark, but is second-last when it comes to raw performance. On the other hand the PowerPC has the best performance but is draining a lot of power that makes it unsuitable for battery-powered embedded systems.

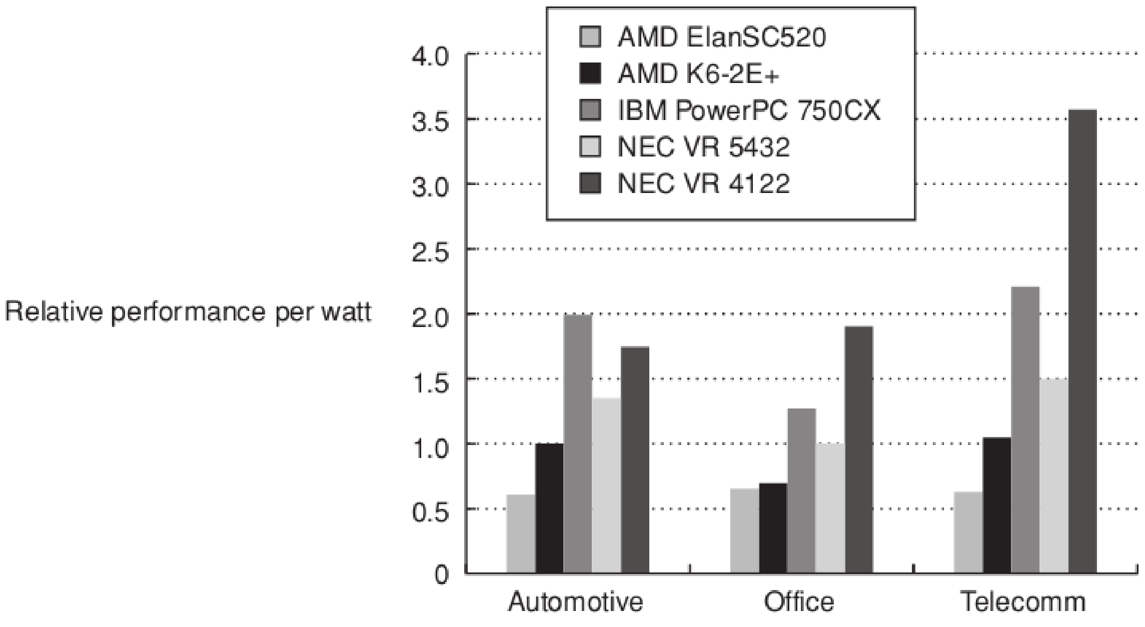


Figure 1.1: Relative performance per watt for five embedded processors [Henn06].

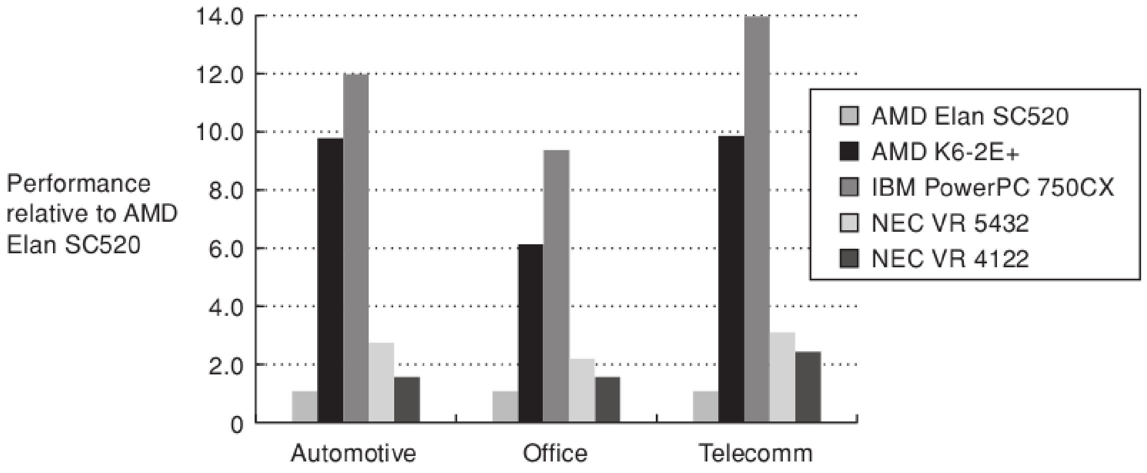


Figure 1.2: Raw performance for five embedded processors, The performance is presented as relative to that of the AMD ElanSC520 [Henn06].

Embedded multiprocessor systems

Nowadays, mainly in servers but also in desktop systems, using multiprocessors is a common way to boost performance. Likewise in embedded systems, multiprocessors are used, with different special-purpose processors. This proves highly effective, since each specialized processor can handle efficiently specific functions. A prime example of an embedded multiprocessor is a modern mobile phone, equipped with one or more ARM cores, several DSPs and even more dedicated ASIC co-processors for specific tasks, like Viterbi decoding[1].

There are also two reasons for the popularity of multiprocessors in the embedded space. The first one is that they make it easier to exploit the parallelism that already exists is many applications, by assigning parts of the code to different special-purpose processors. The second is that, as mentioned before, the parts that are most commonly used are optimized for each embedded processor, and that eliminates the problem of binary software compatibility that still troubles many desktop systems.

1.1.2 Processor design and instruction set architecture

Processor designers today face a complex problem. When designing a processor, they have to determine which attributes are more important, and in general try to design a processor with as great performance as possible while at the same time keeping in line with the constraints in area, power and cost. As

[1]Viterbi decoding uses the Viterbi algorithm to decode a bitstream that is encoded using forward error correction based on convolutional code. It is resource-consuming but does maximum likelihood decoding

mentioned very precisely on [Catth10], *processor design is a game of many trade-offs*, and the designer has to balance the attributes of the processor and many other aspects including instruction set architecture, organization, logic design and implementation. The implementation may include integrated circuit design, packaging, power and cooling. Optimizing the design requires familiarity with a wide range of technologies, from compilers and operating systems to logic design and packaging.

The *instruction set architecture (ISA)* is the most defining part of processor and what defines the communication between the hardware and the software. The main characteristics of an ISA[Henn06] are clearly defined as follows:

1. *Class of ISA* - General or special purpose architecture. Register-memory or load-store oriented.

2. *Memory addressing* - All desktop and server computers use byte addressing to access memory operands. Some ISAs also require that the data has to be aligned to be read and written correctly.

3. *Addressing modes* - They specify registers, constant operands and the address of a memory object.

4. *Types and sizes of operands* - Common examples for fixed-point are 8-bit(ASCII character), 16-bit(half-word or unicode character), 32-bit(word or integer) or 64-bit(double word) and for floating point 32-bit(single precision), 64-bit(double precision) or even 80-bit(extended double precision).

5. *Operations* - Data transfer, arithmetic, logical, control etc.

6. *Control flow instructions* - Conditional branches, unconditional jumps, procedure calls and returns.

7. *Encoding an ISA* - Usually fixed length or variable length. More about this on chapter 3.3.

VLIW and vector processors

Very Long Instruction Word (VLIW) processors are multiple-issue processors that use many, independent functional units to exploit instruction-level parallelism (ILP). Instead of using multiple instructions for different units, VLIW processors have a single very long instruction word that contains multiple operation instructions, one for each of the functional units. Taking full advantage of a VLIW requires to focus on a wider-issue processor so as to maximize the issue rate. In order to fully utilize this architecture, the code generated by the compiler must contain enough parallelism to provide instructions to as many of the functional units as possible in each cycle. Modern compilers use code transformation techniques like loop unrolling or scheduling techniques like local or global scheduling to detect and enhance the required parallelism.

Explicitly parallel instruction computing (EPIC) processors were introduced by HP and Intel to address and overcome some of the common problems of VLIW, mainly in flexibility, code size and improved software speculation. By using explicit indicators for possible instruction dependencies and multiple instruction formats instead of the fixed instruction format of VLIW and so is able to express parallelism more flexibly and reduce the size of the generated code.

Of course, nowadays there is also the choice of *vector processors* for some applications. Vector processors offer operations that can process a lot of data at the same time in special vector functional units, provided that those data are *vector* form, linear arrays of numbers. So vector processors can provide faster results at the same cost but only for specific applications in structured code that the vectorization can be applied. Otherwise, VLIW are preferred for their ability to extract parallelism from less structured code and adapt to all forms of application data.

One of the most common vector architectures used for both desktop and embedded systems is *Single Instruction Multiple Data(SIMD)*, as it was classified by Flynn's taxonomy. SIMD can exploit data-level parallelism, by applying the same operation to multiple data in parallel (vectorized data). It is most effective in applications that show great data-level parallelism, like high performance applications, graphics acceleration and many digital filters.

Orthogonal instruction set

Orthogonal instruction set is a term used in computer engineering to classify an instruction set architecture where any instruction can use data of any type through any addressing. The word orthogonal, meaning “right angle” in greek, is used in a similar way to geometry and mathematics and implies that the ISA provides the capability to move along the operations axis independently of the other addressing mode axis and vice versa, thus enabling all possible operation and addressing mode combinations but forcing a limited set of operational codes and addressing modes.

Many CISC based computers generally follow the orthogonal instruction set, by allowing an instruction to access either the register file or the main memory in several different ways. There are several fully orthogonal computer systems like PDP-11 and VAX-11 or others that are nearly orthogonal like DEC PDP-11 and Motorola 68000.

In RISC architectures, orthogonality is also used, but not full orthogonality because that would lead to a less efficient architecture, and several instruction bits are used instead for other purposes. So there is a trade-off that is usually made for each architecture between orthogonality and enabling other techniques like virtual addresses, longer immediate data or larger register files.

1.1.3 Application specific instruction-set processors

Traditional ASICs (Application Specific Integrated Circuits) have great performance and low energy consumption but lack flexibility since they are designed and optimized to perform a specific task. DSPs (Digital Signal Processors) are flexible and their performance is very good but they are not energy efficient at all. And that is the reason why ASIPs are developed. They perform almost equally well and can also be energy efficient, but their strong point is their flexibility (fig:1.3). The design effort for mapping code on an ASIP is quite low, but still higher than that of an ASIC [Catth10]. Thankfully, there are many automated tools that can help in this and make it less of a challenge.

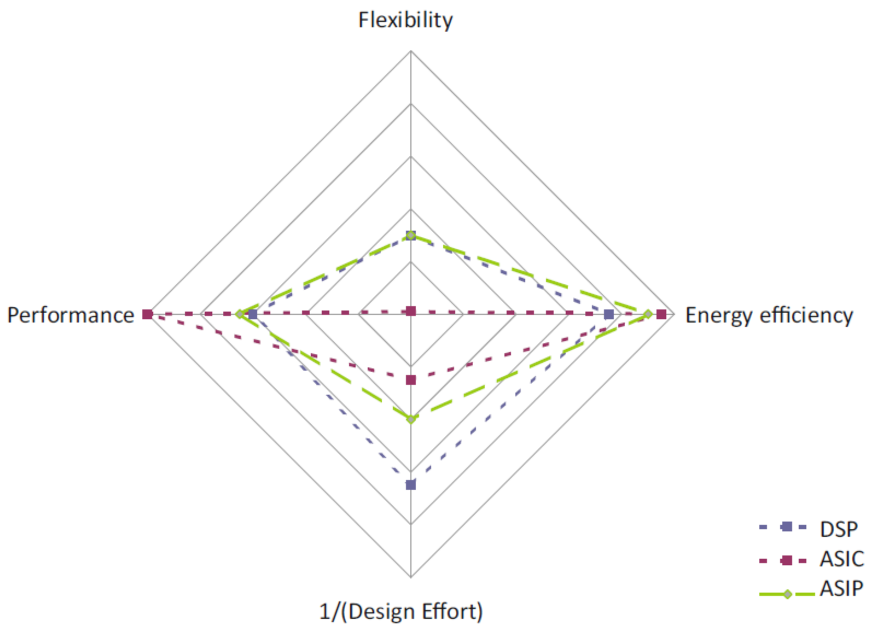


Figure 1.3: Different design styles target different design metrics [Catth10]

Application Specific Instruction-set Processors (ASIPs) are nowadays used increasingly in System-on-Chip design to design a programmable processor with an instruction-set tailored to fit the needs of a specific application domain. They bridge the architectural gap between general purpose processors and ASICs (Application Specific Integrated Circuits) combining the advantages of both "worlds". They are developed using a user-friendly processor description language called *Architecture Description Language (ADL)* (see 2.5), an efficient retargetable C compiler along with accurate simulation and profiling tools, and so considerably decrease the time needed to develop a new processor.

In the application domains of image and video processing, ASIPs can use a combination of VLIW and SIMD along with powerful compilers and other tools and techniques to maximize efficiency and make optimal architectural trade-offs.

Instruction word size against instruction decoder complexity for energy efficiency

Recent results on the power dissipation figures of various latest processor cores show an increasing percentage of the overall consumption is due to the instruction memory and the decoder. As seen both in the ultra-low power biomedical signal processor CoolBio of IMEC,Holst Centre and NXP and ICORE[Zhang08], the power consumption attributed to the decoder takes up 28-42% of the total consumption.

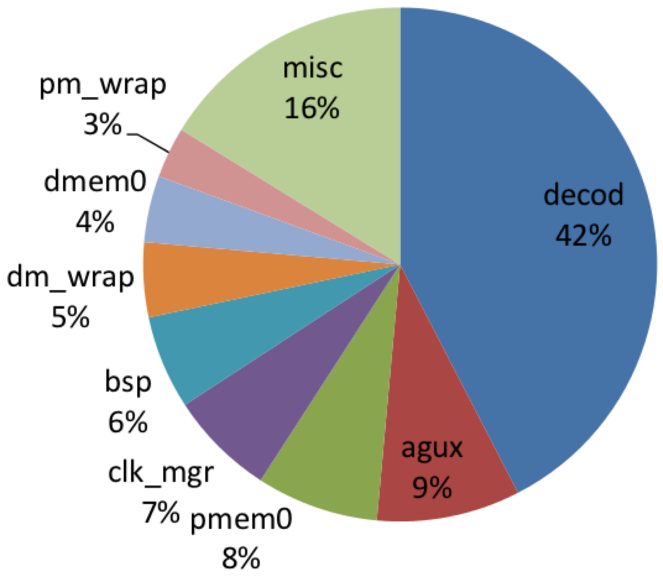


Figure 1.4: Power consumption for different components of CoolBio executing ECG v1.0 code

In commercial processors in the effort to increase their capabilities their instruction set contains not only a lot of instructions compared to the small instruction word, but also these instruction often are very complicated in their structure and addressing modes they have to support.

Concurrently, as noticed on latest commercial SRAM memories, doubling the size of the memory width does not incur an equal rise in power consumption. More about that on a later chapter.

As illustrated in figure 1.5, for a commercial ASIP the average power consumption and the performance required are included in the shaded area of the triangle. For different code optimization level there is also a different different performance required, meaning that better optimized code requires a lower operational frequency, that results to lower power consumption. This means that the average power drops while the mW/MHz for a task rises. If the instruction set is tailored for the functions then the code can be executed faster and more efficiently. Increasing the supply voltage would increase the power consumption and thus the lines of the triangle would rise in the Y axis, at a "steeper" angle. Leakage is omitted for now. Therefore, since bigger frequencies have a negative impact on power consumption and the aim is not simply a low power consumption but a smaller energy consumption per task. So instead of mW/MHz, using J/function is more a accurate metric for energy efficiency.

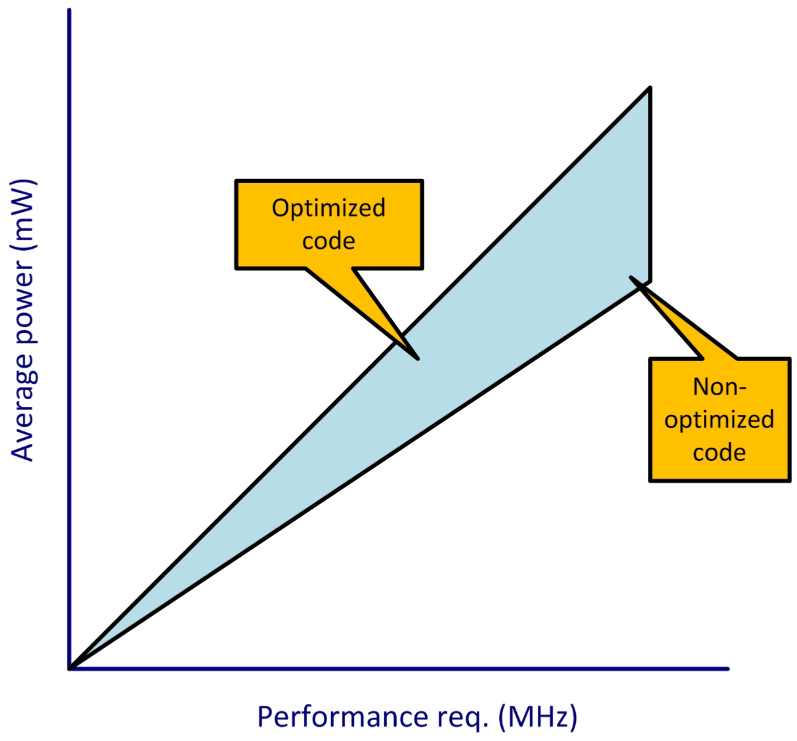


Figure 1.5: Average power and performance depending on the level of code optimization for a commercial ASIP

So it needs to be looked into whether the energy gains from having a longer, simpler and more orthogonal instruction set in an ASIP, or even a fully orthogonal VLIW processor (see fig.1.6) can compensate for the extra energy incurred by the bigger memories needed.

It also the purpose of this thesis to explore the capabilities of a commercial ASIP development tool, like the Target tool suite used for the development of several ASIP cores, investigate the limits of those capabilities and give suggestions for future additions that could be added. This study also looks into how the ASIP tools can be fully utilized to exploit the energy effiency dynamics of a design for a specific application domain.

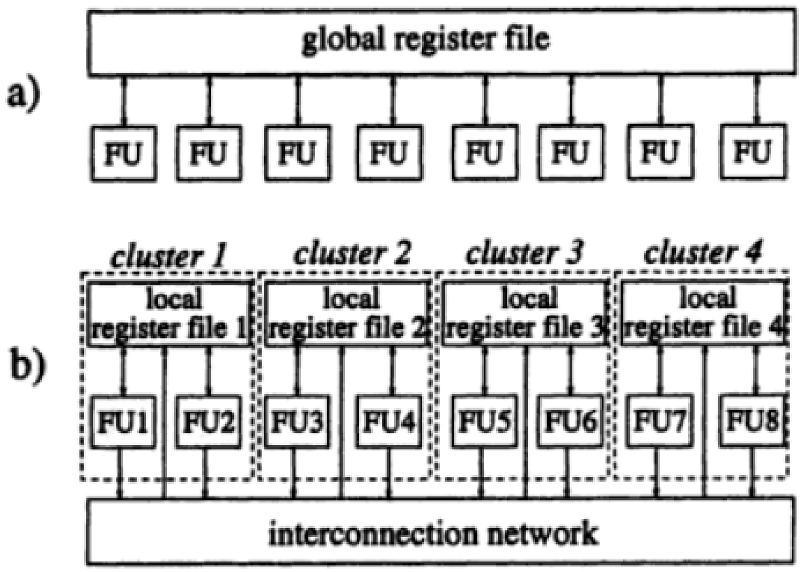


Figure 1.6: VLIW data paths: a) orthogonal b) clustered [Leup00]

1.2 Objectives

The main objectives of this thesis are the following:

- Investigate how different components and overall power consumption are affected by changes in the width of the instruction.
- Experiment with different encodings for the instruction set of an ASIP and notice their effect on the instruction decoder.
- Find the best trade-off between the width of the instruction and the memory size for low power consumption, considering that the former has a very strong impact on the instruction decoder.
- Benchmark and provide suggestions for improving and extending existing designs.
- Automate the whole procedure from designing an ASIP processor to generating the HDL and benchmarking its area and power performance and provide suggestions and feedback to Target

1.3 Thesis Outline

The main body of this report is divided as follows.

Chap. 2 provides related work on the fields of ultra-low power processor design, code compression and encoding and ASIP methodology which are associated with this thesis as well as the background information and case studies that are needed to understand it.

Chap. 3 provides the development framework of Target tool flow, analyzing the way the nML grammar is used to describe an instruction set architecture and tools along with the basic options that they offer.

Chap. 4 includes the analysis and implementation of a SIMD processor and another soft-SIMD and the experimental results of their comparison to a scalar processor and to each other,

Chap. 5 deals with the basic implementation of an orthogonal processor and the various experiments conducted on it to achieve power efficiency. It also shows the results of all the implementations during this thesis.

Finally, Chap. 6 summarizes what's been achieved, provides the conclusions of this study, the open questions and what could be done next.

Chapter 2

Background and Related Work

The last fifteen years there has been a lot of work about ASIPs and various low-power design techniques. This chapter provides an insight to several state of the art approaches to problems and challenges in the field related to this thesis, but also slightly older ones that can be revisited.

2.1 Ultra-low power processor design

Having a an power efficient processor is of grave importance in most systems, but it is difficult goal to achieve because improving power consumption creates a contradiction with other main characteristics of the processor, like performance or flexibility. There are various techniques that can be used for efficient ultra-low power processor design [Piguet06], like CPI (cycles per instruction) reduction, gated-clock mechanisms, optimal pipeline length, hardware accelerators, reconfigurable units and techniques for reducing the leakage power. DSPs are a prime example that helps demonstrate the necessary tradeoff between flexibility and energy efficiency.

The two main constraints for SoC design are none other than power efficiency and computation power. When it comes to the portable consumer market, power efficiency is the defining constraint, particularly in deep submicron technologies, where designers are coming up against new problems, like very low supply voltages, high leakage, long wire delays, networks on chip, signal input slopes, noise and crosstalk effects.

The components that commonly take up a big part of the power pie are the memories. There are various well-known methods for reducing the power consumption of memories (more about that on chapter 2.2), but it should be still kept in mind that the power consumption of the processor itself can also be improved, and that it is the processor that defines the types and sizes of memories needed. But coming up with the *"ideal"* processor, meaning one that is flexible, has great computational power and is also power efficient is virtually impossible. So there has to be a tradeoff between those three, and the best criterion for that would be to base them on the needs of the application of the processor.

2.1.1 Power dissipation

In order to understand how low power techniques work, it is vital to have in mind the various factors that make up *power dissipation*.

Power dissipation is divided itself in *dynamic* and *static* components. Dynamic is attributed to the switching activity caused by temporary current paths (while pMOS or nMOS stacks are partially ON) and charging or discharging the capacitors as gates switch, so it is directly proportional to the switching frequency. The static component is the power dissipated due to static conductive paths between the supply rails or leakage currents, which is there even if there is no switching activity. *Leakage* can be either sub-threshold through OFF transistors, gate leakage through gate dielectric or junction leakage from source/drain diffusion.

A chip and thus power can be considered to be in one of three modes at any time: *active*, *standby* or *sleep*. Active is the power consumed while the chip is working, and is dominated by the switching power. Standby power is consumed while the chip is idle, so if the clocks are stopped it is mostly leakage power. Sleep power is consumed when various components are not needed for a certain time period and their power is turned off to drastically decrease the leakage. However there is an extra cost in energy and time needed to put a component to sleep or wake it up, therefore making this a viable solution only if the component is not going to be used for a long period.

2.1.2 Energy or power focus

The term *power consumption* defines the amount of energy consumed per operation and the heat dissipation of a design, and those two in turn affect several other aspects of the design, like battery life, cooling, placement and packaging. Therefore, power dissipation plays an important part in the design.

Many times, the power characteristics of a chip are described with power for a set frequency, i.e. 10mW @ 1GHz. It is easy to calculate the energy but reporting an energy number makes things much more clear and helps avoid misunderstandings.

A common misunderstanding in embedded system design on whether a designer should aim for energy or power optimization, considering that the two are relative to each other (Energy = Power x Time). But as shown on figure 2.1, there is a distinct difference between energy and power, considering that power is the instantaneous power in the device but it is energy (the area under the curve in the figure) that is truly important for portable systems and actually determines the duration of the battery.

In high performance systems, where the basic focus is performance, there is a consumption limit set by the technical characteristics of the chip and its cooling capabilities. However, in portable and embedded systems it is of great importance to keep a low *energy* limit, because the focus lies in the maximum number of computation in the time range between battery charges. So, in portable and embedded systems having a system with 50% less power consumption that is compelled to run on 50% its frequency (to keep the power consumption at low levels) leads to exactly the same energy, since the total energy consumed for the same task that runs on 50% "low-power" but takes twice the time to complete is still the same. So low power consumption by itself is clearly not enough.

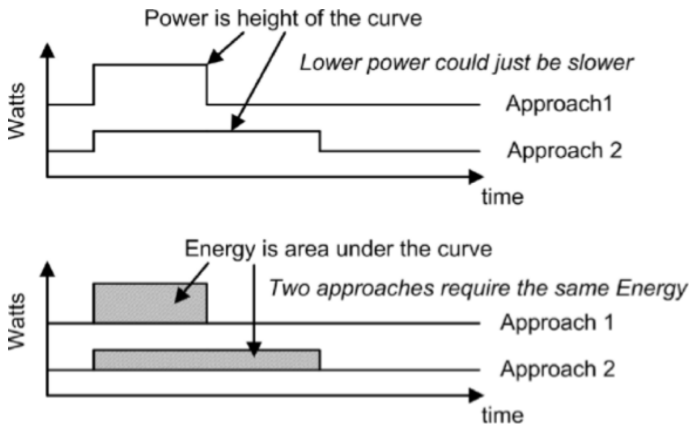

Power is height of the curve
Lower power could just be slower
Watts
Approach1
Approach 2
time
Energy is area under the curve
Two approaches require the same Energy
Watts
Approach 1
Approach 2
time

Figure 2.1: Power versus energy [Keat07]

2.1.3 Low energy metrics

In literature normally the power efficiency of processors is indicated with figures of merit like mW/MHz or pJ/cycle describing the energy cost of processor cycles. It is interesting to have a low number since that shows the low-power properties of a processor core. It shows the power dissipation at a predetermined clock frequency. Using this figure of merit disguises however the more important aspects of processors for wireless sensor nodes (and other battery-operated) devices. The amount of energy to do a certain job is a more important metric. What really needs to be optimized is the amount of energy a task consumes. Therefore, an important figure of merit would be Joule per task, irrespective of the number of clock cycles (or clock frequency) a processor core needs.

Another popular way to calculate the performance to power consumption ratio is in MIPS/w (Million Instruction per Second per Watt) or w/MIPS respectively. Using this figure it is relatively easy to estimate the energy consumption for a given application or task.

A very important motivation that explains the reason why DSPs and ASIPs were introduced is due to the large variation in instruction sets. Processors with large data paths typically need less cycles to complete a task, at a higher J/cycle cost, but typically consuming less energy for the job. For low energy design this also is beneficial.

Processor optimization often leads to complicating the instruction set extensively to keep it small. However the introduced overhead in instruction decoding can potentially lead to increased area and energy.

2.1.4 The deep sub-micron era

Technology is advancing rapidly, and that has a strong impact on the relative merits of different circuit techniques, and ultimately the way designers handle them, with regard to the future. For example, gate delays are improving way more rapidly than the delays of the interconnecting wiring, and threshold drops are becoming more dominant of the supply voltage. Also leakage is increasing. A designer needs to be aware of impending changes like these to ensure the continuity of his creations.

With the increasing complexity of the digital integrated circuits, it is anticipated that in future smaller scale technologies the problem of energy consumption will only get worse. Lower supply voltages are becoming more attractive, because reducing V_{DD} has a quadratic effect on the dynamic power consumption, assuming the same clock rate is sustained.

Many nanometer processes have now reached a point where it is no longer possible to design a high-performance chip without paying attention to its power consumption, because high power consumption results to high heat output and that might prove impossible to cool. Thus, in modern systems designed for speed that use extra logic to be more efficient, a common method is to simplify them. So, if for example a core can be simplified in order to have 80% of its performance for only half of the power consumption, then we can use two cores to have 160% of the performance for the same power consumption.

Another common problem is that many designers are accustomed to focusing on dynamic power. But *leakage* in its various forms (see 2.1.1) is becoming increasingly important in nanometer processes. Failing to account for that can very well lead to much higher than expected consumption and also functional failures in the more sensitive components.

2.1.5 Performance to power consumption ratio in different processor types

As shown on figure 2.2, there are many different architectures that can be used for the same purpose. The more specialized the architecture, the better the performance and power efficiency but less flexible the resulting processor. So for example we can use a general-purpose microprocessor with reduced performance but high flexibility or a custom ASIC with high performance and no flexibility whatsoever. The same task on a processor of different level, can have great variance in execution time. Therefore the right processor, or co-processor should be picked to handle each task efficiently. For example, the number of clock cycles executed for the simple task of a counter can vary from one cycle in a hardware counter or several instructions with each one requiring many clock cycles each.

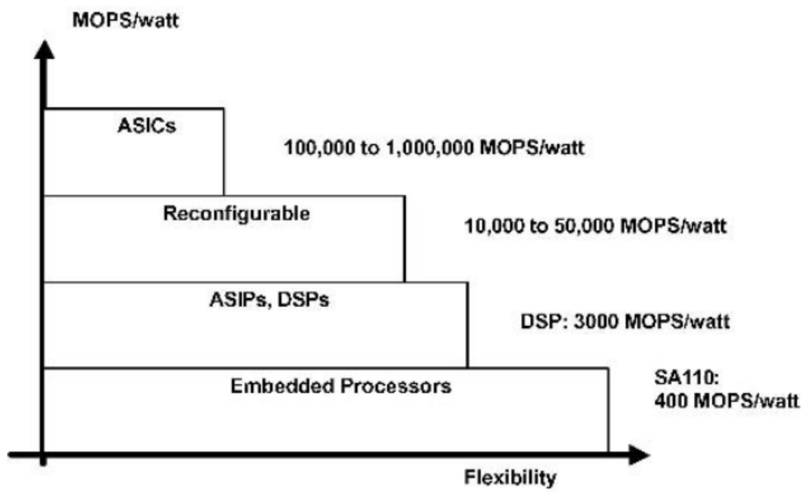


Figure 2.2: MOPS/watt versus flexibility [Piguet06]

But apart from those two extremes, there are several solutions in between, mainly reconfigurable processors and the aforementioned DSPs (chap. 1.1.1) and ASIPs (chap. 1.1.3). Reconfigurable processors prove to be very useful, as they allow the configuration of specialized instructions and execution units to the specified application.

The rest of the parameters that need to be defined is matching the *data width* of the processor (and subsequently the memory) to the required data. The required data does not necessarily need to be the same as the processor, so it is possible for example to execute 16-bit data on a 8-bit processor, but there is an extra cost and increased execution time for that. Additionally, each processor performs considerably better when facing the task it was developed for, and no processor is best for everything. For example, a DSP processor is much better than a microcontroller in performing a filter, but the microcontroller can handle control operations more efficiently. That is why we usually use a microcontroller with several dedicated co-processors to handle everything properly. This way each task is executed by the smallest and most energy-efficient component, but rarely all of them are working at the same time in parallel.

2.2 Memory efficiency

2.2.1 SRAM

A SRAM is a memory cell array consisted of SRAM cells that are able to read, store and write data for as long as the power supply is on. A common 6-transistor SRAM cell(fig2.3) can be an order of magnitude smaller than a flip-flop. This 6T cell is compact, requiring less wiring and so features a small dynamic power consumption.

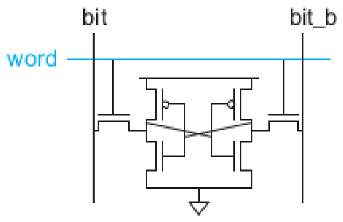


Figure 2.3: A typical SRAM cell composed of 6 transistors (6T)[Weste11]

The cell contains a pair of weak cross-coupled inverters holding the state and a pair of access transistors to read or write it. In order to write in the cell the desired value and its complement are driven to the bitlines, bit and bit_b and then the wordline, word is raised. This way bit or bit_b are pulled down to indicate the new value. The challenge in SRAM design is have as small an area as possible but ensuring that the state is stong enough to withstand the influence of leakage and keep the state during a read, but weak enough to be overwritten during a write.

SRAM cells are structured in memory arrays of m address lines and m data lines. So the size of an SRAM is 2^m words, or $2^m \times n$ bits. Memory cells can have one or more ports for access. These ports may be read-only, write-only or support both, but not simultaneously. For larger SRAM memories, multiple smaller arrays are combined so that the wordlines and bitlines can be fast, narrow and low-power.

2.2.2 Memory power efficiency

There are various well-known techniques for low power consumption of memories. That usually includes cutting the memory in small pieces and only one piece is addressed to fetch or to store data (cache, hierarchical, divided workline and divided bitline).

In the quest to optimize processor cores for energy and performance, somewhere a bandwidth limitation of memory is hampering further optimization. Therefore often multiple, local scratchpad, memories are used. We have found that memory consumption may easily become a power dominating component. If there is a strongly memory-intensive application, the memory accesses can go over 50% or even 60% of the total power budget of a typical DSP processor. As a result most DSP processors try to exploit the memory hierarchy and the register file in an effort to reduce the memory acceses. Pre-fetch techniques are particularly effective in DSP designed for applications with large data objects. Code density is also very effective, provided that the overhead of encoding and decoding makes up for the benefits.

The general guidelines followed to reduce memory consumption are turning on only the necessary subarrays to minimize the dynamic power, keep the other subarrays in sleep mode to minimize leakage and reduce the voltage levels to a the minimum required for the memories to function without loss of data or vulnerability to interference.

Modern SRAM memories have developed a lot during the last decade. As we can see in figure 2.4, featuring slightly old generation (2008) SRAM memories at 90nm, larger memory block are more area-efficient than smaller ones[Katev11]. Also, the difference between the different word sizes in area starts with a big overhead for the smaller word sizes, but become negligible in larger block capacities.

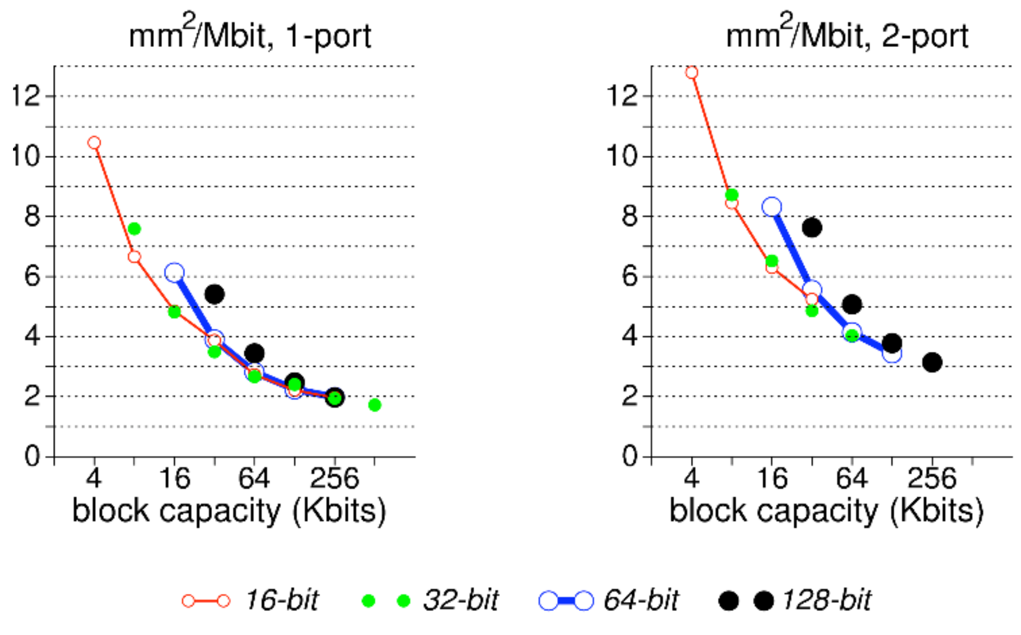


Figure 2.4: SRAM area needed for different block sizes[Katev11]

Figure 2.5 shows the power to performance ratio for different block capacities. Power consumpion is proportional to access frequency (μW/MHz). It can be observed that for every single one of the word sizes, the power to perfomance ratio rises slowly for the first few block capacities (due to increasing word-line and bit-line capacitance), but as the block capacities increase the average power increases at a much faster rate (because at some point more sense amplifiers are required). It is also notable that the gap between the power consumpion of each of the word sizes is getting bigger for larger word sizes and for larger block capacities as well.

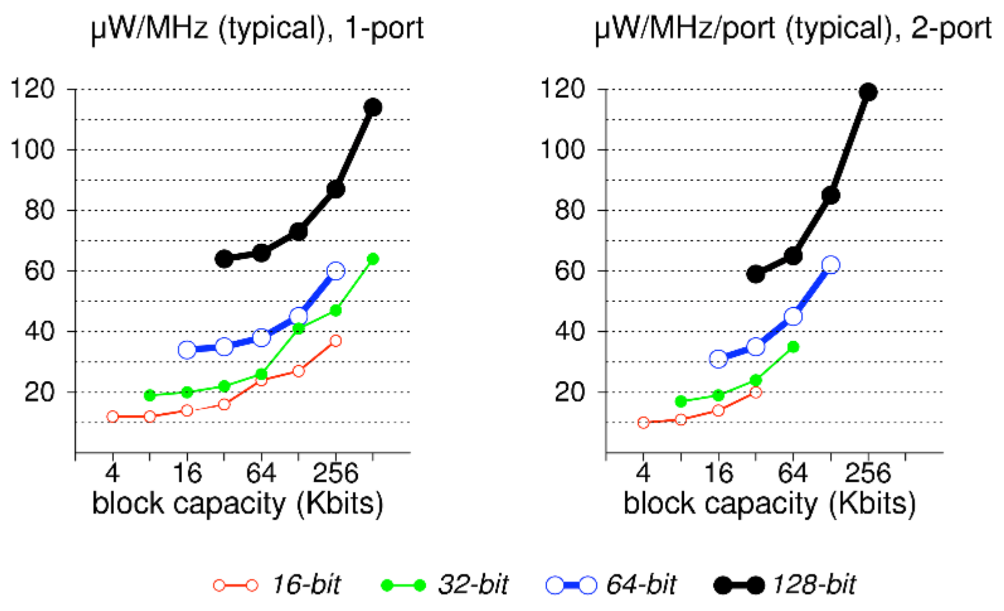


Figure 2.5: SRAM Power consumption for different block sizes[Katev11]

There were several measurements taken on the commercial memories available (fig.2.6) for different word sizes of 2^5 to 2^{14} and data width sizes of 16 to 80 bits. We can see that it is possible to double the memory size (and also the instruction width) without also having a double power cost per memory access.

This shows that that bigger memories can used and despite the extra power cost, the total design can benefit if this helps the rest of the components of the processor.

2.2.3 Memory addressing modes

For specific purpose processors like DSP, there needs to be a well defined set of special addressing modes. These assist the processor in handling special data types or large data in less clock cycles, so this results in less energy consumed for the same application.

The design has to balance the benefits of the extra addressing modes against the extra complexity they introduce. They can be especially effective when it comes to FFT and other similar computations, however there needs to be careful planning of both the addressing and the software stack support if they are to provide efficient data structures that minimize the use of memory.

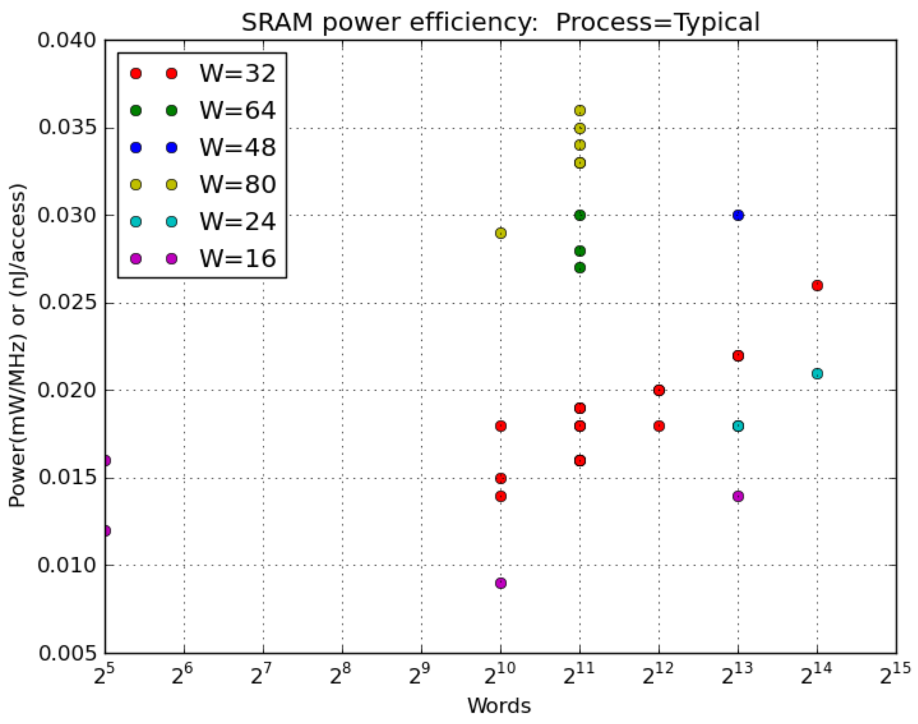


Figure 2.6: Power consumption for different word and block sizes in commercial SRAM memories

2.2.4 Loopbuffers

Loop buffering[Barat03] is an effective scheme used to reduce the energy consumed in instruction memories. In many application domains, *eg.* multimedia, a great part of execution time is spent in small program segments that repeat. Loop buffers are employed in that case to store a small number of instructions, so as to avoid the relatively much more expensive instruction cache.

2.3 Techniques for energy-efficient processors

There are several techniques that can be applied on a design for power reduction, and each one of them is usually most effective on a certain level [Piguet06]. Most of the gain in dynamic power can be saved at the highest levels (see table 2.1). At the *system* and *architecture levels* partition, activity, number of steps, simplicity, data representation, memory hierarchy (cache, distributed memory or centralized memory) and locality. But these choices depend to a great extend on the application. At the *circuit level* the techniques typically focus on dynamic power reduction with methods like gated clocks, pipelining, parallelization, very low Vdd, several Vdd, variable Vdd (DVS or dynamic voltage scaling) and V_T,

	Dynamic Power	Static Power
High-level	Reduction of the number of executed tasks, steps and instructions. Processor types. Processor versus random logic. Reconfigurability	Remove units that do nothing or nearly nothing
Architecture	Asynchronous Encoding, Parallel Pipeline, Simplicity	Architectures with less inactive gates
Circuit Layout	Gated Clock, Sub 1V, DVS, Low V_t, Low-power library and basic cells	Gated Vdd, MTCMOS, VTCMOS, DTMOS, stacked transistors
	Activity reduction, Vdd reduction, Capacitance reduction	

Table 2.1: Power reduction techniques [Piguet06]

activity estimation and optimization, low-power libraries, reduced swing. At the *logic and layout levels* it is of great importance to choose the right low-power libraries and the right mapping method. Finally, at the *physical level*, there is the choice of layout optimization and technology. Especially in deep submicron technologies where reducing leakage and static power becomes more difficult, an effective low-power design has to address all design levels.

For an architectural design strategy for low power to be truly effective, it must be *holistic*. Every part of the system needs to be analyzed and designed to be power efficient and fit to each other perfectly. A careless design could lead to a few components with greatly reduced power consumption but increased in many of the other components, therefore a system with an overall worse power consumption.

2.3.1 Low-power techniques in circuit design

There are various techniques that have a beneficial effect to general purpose processors and consequently to ASIP's as well. Those techniques cover a great range from low-level techniques like voltage scaling and clock gating to higher level ones like code compression algorithms and scheduling optimization.

Since an ASIP is translated in a HDL as a complex finite state machine where the state transitions are triggered by the input data and the ASIP software, most of the known techniques for energy efficient hardware design can also be applied accordingly.

While it is not the purpose of this thesis to analyze extensively low level circuit design, the system designer needs to take them into account, so that they can later be efficiently applied.

Dynamic voltage and frequency scaling

There are many different applications that may execute on a system but each one of them has different performance requirements. Additionally, when it comes to embedded systems there are also power requirements. For example, running Matlab requires much greater performance than playing Minefield. In these systems a very useful technique is employed called *dynamic voltage scaling (DVS)* or *dynamic voltage/frequency scaling (DVFS)* [Burd00]. The aim of DVS is to decrease the power consumption by exploiting the time-varying computational load of different applications in embedded systems and adjusting the performance of the system to match the needs of each application when that particular one is being executed (fig 2.7). Then the supply voltage is reduced to the bare minimum required for the set frequency. As a result, there are significant gains in power consumption by up to a factor of 10x without sacrificing peak throughput.

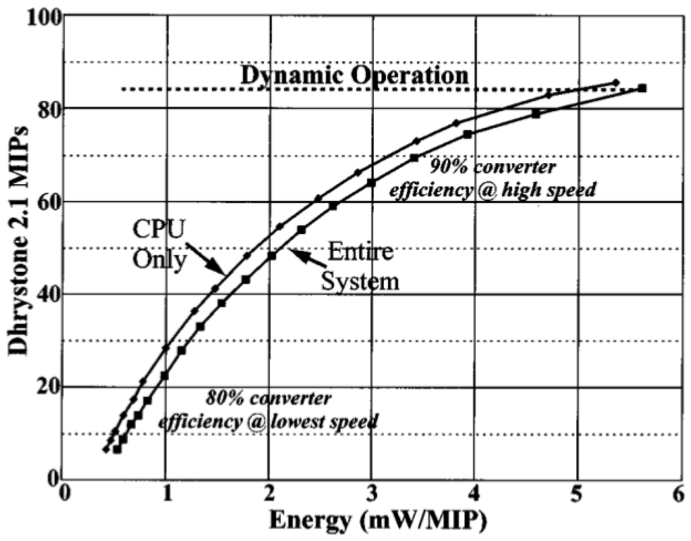


Figure 2.7: Measured throughput versus energy consumption [Burd00]

DVS can also prove useful for reducing leakage during periods of low activity, because sub-threshold and gate leakage are strongly sensitive to the supply voltage.

Clock gating

Clock gating is a design methodology for reducing ASIC power consumption by inserting enable signals before the clock signal of a block. This way when the block is not used it is turned off using the disable and the switching activity of the registers is halted. That reduces the dynamic power consumed to zero but incurs an overhead for the extra logic required. This technique is fairly simple as long as we don't add to the critical path of the design, and provides great gain in systems when we have idle parts of the design for a long time.

Power gating

One of the biggest problems as the CMOS technology scale keeps getting smaller is the problem of static leakage. A widely used technique to reduce static current during sleep mode is to turn off the power supply to the sleeping blocks, known as *Power Gating*, originally proposed as Multiple Threshold CMOS (MTCMOS) [Mutoh95]. In order to accomplice this, special sleep transistors (fig 2.8) are introduced in the design that connect V_{DD} and V_{DDV} when the header switch is ON or cuts off supply to V_{DDV} through V_{DD} when the header switch is turned OFF. In the latter case, as V_{DDV} gradually sinks to 0, the output of the block may go to unwanted voltage levels, so the isolation gates are needed to force the outputs to a valid level during sleep.

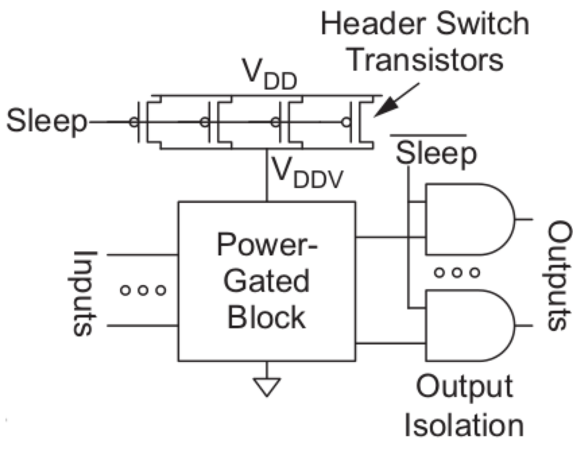


Figure 2.8: Power gating [Weste11]

Activity and optimal total power

In systems with processors that have low or very low activity, the ratio of dynamic to static power is expected to be really small, because the gates in the system consume a set amount of static energy while total switching activity is quite low. Leakage can be considered roughly proportional to the number of gates and the duration of the clock cycle. Since what we want an optimal total power, there it is worth looking into whether making better use of a smaller amount of switching transistors or gates can prove helpful. However, that by itself is not enough. Because having a smaller architecture has a negative impact on performance in some cases. so to cope with that the supply voltage is increased and the V_T

decreased, resulting in increased total power. So, to actually accomplice the aim, one would need to take into account various factors, mainly the speed constraints and the logical depth (LD). Usually this method can be best applied in pipelined architectures.

2.3.2 Low-power techniques in architecture level

CPI reduction

Most well-known 8-bit microcontrollers are based on CISC (Complex Instruction Set Computer) architectures, in which every instruction format contains several bytes and several memory accesses are required for the execution of a single instruction. This results in a CPI (clocks per instruction) of values 4-20, and according to the formula that calculates the performance of a processor in MIPS (millions of instructions per second), MIPS = $f/$ CPI, in order to achieve a high performance the processor would have to be clocked in a high f frequency, because dynamic power is proportional to the frequency that results in a higher dynamic power consumption. In contrast, RISC architectures may have a similar amount of transistors but can offer a CPI of almost 1 (usually a bit higher than 1), therefore offering greater energy efficiency per instruction executed. CPI reduction is the technique that can bring most of the improvements in a single processor system.

Gated-clock mechanisms

Clock gating as explained in section 2.3.1, can be applied in various components of a design (or certain parts of those) in a way that allows the designer to disable the clock and thus the transitions and switching in those parts. For example in the 3-stage pipeline CoolRISC core it is used on parts of the ALU and also for the instruction register. As a result, a branch is only executed in the first pipeline stage and there are no costly (in energy) transitions in the second and third stages of the pipeline. It is obvious that gated-clock mechanisms can be used in conjunction with pipelined architectures for better power results. Many modern CAD tools support automatic gated-clock mechanism insertion.

DFF versus Latch based design

One of the main problems of today's processor design is *clock skew*. It is extremely difficult to design a clock tree with the smallest possible clock skew while avoiding possible timing violations. This problem has been augmented in deep submicron technologies, considering that the smaller the technology the larger the wire delays as compared to gate delays. Most modern processors make use of a single-phase clock and are based on D flip-flops, and in deep submicron technologies the problems of reliable clock input slopes and clock input capacitance in standard cell libraries are all parameters that have to be satisfied for a successful design.

A possible solution for these problems could be replacing the conventional single-phase clock comprised of DFFs with only *latches with two non-overlapping clocks*. This clocking scheme proves to be more reliable, more robust to clock skew and less prone to low voltage timing violations than a DFF based one. In order to maintain such a clocking scheme, two clocks are needed, each with twice the frequency intended for the whole system. To ensure there are no timing violations, the clock skew of each one has to be smaller than half the total period.

This scheme is generally more energy efficient because in DFF-based systems there is a main clock tree with large costly capacitance while in latch-based there are two smaller ones with smaller capacitance but more relaxed timing constraints that more than make up for it, resulting in a smaller total power consumption. Latch based clocking can also be used to verify the chip functionality and detect design problems because it eliminates clock skew problems, which are much harder to accomplice in DFF-based designs. Latch-based design also enables *time borrowing* and makes use of *timing barriers* that stop the propagation of the clock signal and halt the transitions, thus reducing power consumption. It also brings a significant reduction in the number of MOS transistors needed and the total area required because it allows the master part of the registers to be common for all registers.

Combining latch-based design with clock gating in a pipelined architecture can greatly reduce the total power consumption, by clock gating each stage of the pipeline containing a latch register with individual enable signals. This reduces the number of transitions in the design compared to a DFF-based design (see figure 2.9), since each DFF is equal to two latches clocked and gated together. Finally, latch-based design allows for safe clock gating methodology without glitches in the clock or the need for memory elements (as is in DFFs). It is estimated that for all of the above reasons, latch-based design bears an improvement factor of 2 over a similar design with DFFs.

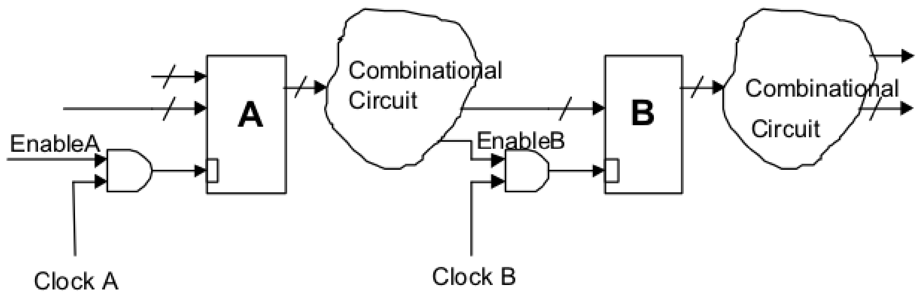


Figure 2.9: Latch-based clock gating [Piguet06]

Optimal pipeline length

The number of the pipeline stages that can prove ideal for a processor depends on the type of the processor and the targeted application domain. A pipelined microprocessor is designed to execute N-cycle instruction overlapped in a N-stages pipeline. In each cycle a new instruction enters the pipeline, another one is completed and the rest just move to the next stage. Ideally this would result in a CPI of 1, if it wasn't for the hazards that cause pipeline stalls.

The most common and unavoidable type of hazard is branch hazards that occur when the target instruction after a branch is determined in a later stage of the pipeline. So the pipeline is filled with NOP bubbles until the address of the next instruction is settled. There are various methods to work around this problem. Either using a branch delay slot that always executes the following instruction after a branch, thus reducing the number of lost cycles by one, using a bypass technique to forward data to a preceding stage before they would normally be available or finally using branch prediction techniques that attempt to predict the verdict of conditional branched. Taking into account that more often than not computation intensive programs contain a lot of loops, it would be safe to assume that in those programs the most common case for a branch would be for it to be taken, so that the instruction address is back to the first instruction of the loop. Of course there has to be a safety mechanism that prevents the instructions from committing their changes if the prediction was wrong. Another prediction scheme based on statistics would be to assume that the branch will behave in the same way as it did last time.

In general pipeline hazards can be solved in a variety of ways, following different kinds of approach. In a static approach the compiler is responsible for reorganizing the code and inserting NOP instructions. In a dynamic approach the processor hardware is in charge of solving the pipeline bubbles at run-time. If out-of-order execution is a supported then the code is reorganized dynamically. Other possible approaches include pipelined multi-thread architectures or a short pipeline (like CoolRISC) where branch instructions are executed in a single cycle.

Whether or not any of the above approached are needed or can be applied depends on the depth of the pipeline. The need for increased performance has increased the depth of pipelines in microprocessors from three to five, six or even eight stages. The more the stages, the better the performance, but the downside is that we have to employ several of the aforementioned techniques to keep the performance to power ratio to a healthy level.

Dedicated DSP cores, multi-core systems and multi-threading

General microprocessors are not very efficient with digital filter code. For that reason support for special DSP instructions was added to the instruction set of a processor. But this method is not very effective and causes quite an overhead for the processor. Cores having control and DSP instructions result in lower performance, higher power consumption, approximately a 30% reduction in maximum frequency and 30% increase in area required. That is why processors designed and optimized exclusively for DSP tasks were developed. But they in turn cannot execute control instructions efficiently. The solution for this came in the form of multi-core system-on-chip, where control instructions are handled by the microcontroller and DSP instructions but the DSP co-processor. Multi-core systems claim a better performance to power consumption ratio, less heat dissipation and smaller area required.

DSP architectures have many advantages. They are dedicated for executing arithmetic operations and are very energy efficient in DSP algorithms. They can complete several memory accesses in a single clock. They can fetch an instruction from the program memory, fetch the operands and store a result in a single clock cycle. Their memory organization is either the classic load/store used in processors for increased parallelism or they are designed to fetch directly the data they need from the memory for a simpler design. Many of them use specialized addressing modes to address two data RAM through two banks with different points for the unprocessed and the processed data and circular addressing (for modulo). This way they can apply a DSP algorithm on arrays of data in an efficient way. Finally, DSPs can be easily optimized to execute loops with zero overhead. For all of the above reasons, dedicated DSP cores show increased performance and can execute a DSP task in a fragment of the time a normal processor would.

To move a step further from multi-core systems, multi-threading is also employed as a way to enhance the capabilities for parallel execution of instructions by allowing threads that are independent to each other to execute on different functional units of the system. Multi-threading is used on high-performance systems but requires code that supports parallelism for it be fully utilized.

It is yet unclear but a multicore system containing an array of identical parallel DSP cores could be the answer to leakage increase [Piguet06]. Pushing frequencies too high implies lowering the V_T, which results in increased leakage power. To counter that, arrays of identical DSP cores with high V_T could provide the same computational power with less leakage. The result is high computational power whereas the total register count and therefore the leakage is kept small.

VLIW DSP cores

There are also DSP cores in VLIW architectures, e.g. in commercial processors from Texas Instruments, like the TMS320C6X containing an eight-issue VLIW 16-bit fixed point processor, that also includes two MAC units and six ALU, in which a maximum of eight instructions can be executed in parallel wth 32-bit data. Some implementations also include hardware accelerators for Viterbi or Turbo decoding, which leads to great performance results.

However, the peak performance of these implementations is impossible to reach due to the nature of the code that can usually be broken to no more than three or four parallel instructions, instead of the eight that are needed to fill all of the execution units. Even if they are indeed filled, the energy required to fetch a 256-bit VLIW instruction word from the program memory is way bigger compared to the cost of a 32-bit instruction word in a superscalar DSP core.

Superscalar DSP cores

Superscalar DSP cores contain multiple MAC execution units, much like the VLIW DSP cores, but unlike them they take advantage of their small 32-bit program data size. The problem in these cores is the increased area size and consumption of the decoder, because a big number of complicated instructions with different addressing modes have to fit in the instruction set.

Reconfigurable DSP cores

Reconfigurable DSP cores lay in the area between the large power consuming FPGAs (Field Programmable Gate Arrays) and the programmable (but not reconfigurable) DSP cores. They can be reconfigured at the functional level and their interconnections can also be rearranged. Reconfigurable cores resemble FPGAs in many ways, but have way less potential for reconfigurability. However, they can be way more power efficient by allowing the reconfiguration of a small number of execution and addressing units. This way, the power dissipation of the operands fetch is minimized and the addressing modes can be optimized for a given DSP task. As a result, it is only the operators of the functions units in the architecture that end up taking up most of the power consumption.

It should be taken into consideration that the price to pay for a reconfigurable DSP is the power consumption of the reconfiguration bits added in the configuration registers of the special hardware. This reconfigurable hardware is necessarily more complex in terms of transistor count and thus more power consuming. Reconfigurable DSP cores also have to deal with software issues as users can define new instructions or addressing modes but the development tools cannot always support them in an efficient way.

2.4 Code compression and encoding

Data compression is a cost-effective way to increase the throughput in communication bandwidth or utilize storage capacity without significant overhead. It removes redundant information inherent in the original data or simply uses less bits for the same data, thereby enabling a communication link to transmit the same amount of data in fewer bits. For storage systems, fewer bits are actually stored thus increasing the effective storage capacity. There are many compression algorithms, but for better results there are specialized algorithms that can be used for specific tasks. Several data compression techniques have been implemented in either software or hardware. However, software implementations are not able to cope with the high requirements in high-end systems so hardware implementations are used there. They are fixed hardware and cannot be customized. As a result, it has to be parameterizable and take into consideration resource constraints, speed of operation and compression ratio.

In processors and embedded systems having an effective way to compress code can prove really helpful, because apart from helping utilize the memory better, using compressed code and therefore a reduced size executable can also affect beneficially the size and the power consumption of the system. Furthermore, having a more compact program compared to a non-compressed one reduces the cache misses for the same cache or allows the use of a smaller one. That is particularly important, as in modern systems the gap between processor and memory responses is growing, making the memory response a bottleneck, especially in large memories that require longer time to respond.

However, to make sure that using compressed code can truly be beneficial for the system, the designer has to account for the extra cost of the encoding and especially the decoding of the code. The decoder of the instructions can be positioned in different places in the system, thus having a different impact depending on that place. For example, it can be positioned between the cache and the main memory or between the cache and the instruction pipeline.

Unlike normal data compression, *instruction data compression* requires a different approach. Achieving maximum compression but making the encoding and decoding schemes really complicated, could sometimes result to making the decompressing component the bottleneck of the whole system.

There are various methods proposed for efficient instruction code compression. Some of the most common are *Huffman coding*, *dictionary-based* methods, *statistical-based* methods or various combinations of those [Beni02]. There have been also many notable industrial attempts, including ARM's Thumb[ARM],[Xu04] and MIPS16 [Kiss97].

There are also methods that can be used for flexible VLIW architectures[Xie07] or similar with higher compression using LZW-based compression in [Lin04].

Huffman encoding

Huffman encoding [Huff52] is a widely used and very effective technique for compressing data, with gains of 20% to even 90%, depending on the characteristics of the code being compressed [Corme01]. Since any data for compression is considered to be a sequence of characters, Huffman's greedy algorithm using a sorted table containing the frequency of occurrences of each character, comes up with an optimal way to represent each character as a binary string.

To solve the problem of optimal binary character coding, where each character is represented with a unique binary string, the algorithm uses a variable-length code instead of fixed-length. In that variable length-code, the most frequent characters have priority and are given short codewords while infrequent characters are given longer ones.

A simple example of Huffman coding can be seen in fig. 2.10. The first tree shows the initial data and the second is the final result of the coding. Each leaf is labeled with a character and its frequency of occurrence. Each internal node is labeled with the sum of the frequencies of the leaves in its subtree. (a) The tree corresponding to the fixed-length code a = 000, . . . , f = 101. (b) The tree corresponding to the optimal prefix code a = 0, b = 101, . . . , f = 1100.

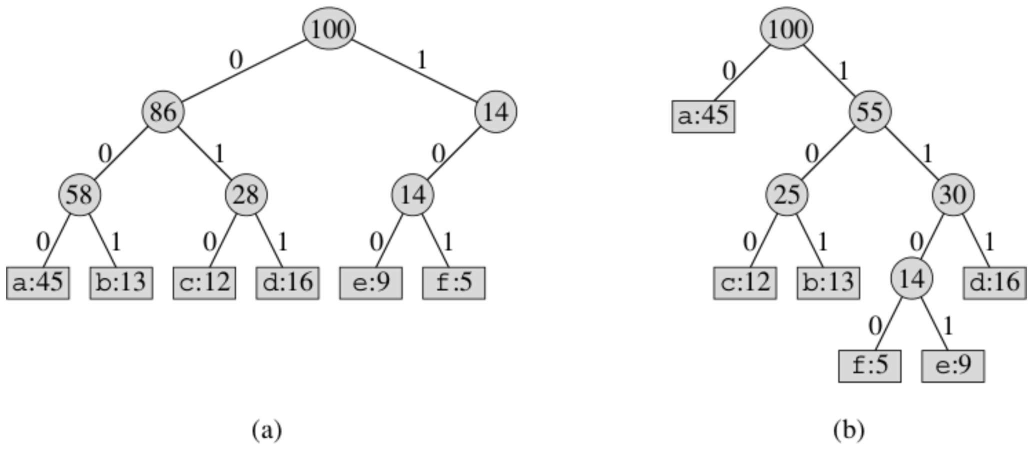


Figure 2.10: Trees showing the initial data and the results of a simple Huffman coding. [Corme01]

The basic notion of the Huffman code can be successfully applied in instruction-set encoding, but it is impossible to take full advantage of the variable-length code in the same way. Nevertheless, it helps a lot in reducing the complexity of the instruction decoder, by allowing the designer to substitute '0's and '1's with don't care bits ('x's).

In one of the early Huffman-based encoding schemes [Wolfe92], the CCRP (Compressed Code RISC Processor) was introduced. The basic aim of this processor was to compress the code so that the processor sees fixed-size, easily decoded instruction that can keep the pipeline full and can potentially provide support for an implementation that enables the execution of multiple instructions per cycle, with a simple addiction of a new cache design. The researchers chose Huffman encoding over others for combining simplicity and effectiveness, providing an optimal encoding for a fixed size input alphabet, but they also modified it a bit to improve its performance. The results showed considerable gains, especially for slower memories. So the processor could benefit from the dense code, with the cost of a small performance overhead.

Recent implementations using Huffman code[Bonn08] combine it with statistical or directory schemes for more efficient compression of the data.

Markov modelling

Many mathematical systems have the property that given the present state, the past states have no influence on the future. This property is called the Markov property and systems that have it are called *Markov chains* [Hoel72]. A Markov chain is the most simple example a *Markov model* and is characterized by random variables that satisfy the Markov property and have stationary transition probabilities. Markov chains are worth looking into because they can model a large number of mathematical systems, hence have a large number of applications in many fields.

In [Corma87] a minimum-redundancy code algorithm is used to describe a message generated by a Markov chain model. Along with an adaptive coding implementation of Huffman code or Ziv-Lempel, the resulting Dynamic Markov Compression (DMC) performs quite well compared to earlier techniques.

Other publications[Hatt95],[Leka99],[Maha05] take Markov chains a step further with *Semi-adaptive Markov Compression*. In [Leka99] there is a very good application of an arithmetic coding and instruction compression framework based on the Markov model. It allows a processor to decompress and use the compressed code during runtime. The results, as the suggested architecture was tested on Analog Devices Sharc and ARM's Thumb show average compression ratios of 41-48% for Sharc and 56% for ARM (outperforming Thumb's 68% ratio). Compared to other implementations, SAMC shows superior compression performance over all algorithms except Semi-adaptive Dictionary Compression (SADC)[Leka98]. In terms of speed it cannot match the fast dictionary coding methods, but it can perform comparably to a Huffman decoder. In area, the requirements are a bit bigger than a Huffman decoder. So, if the dominant design requirement is decoding speed, the dictionary methods are preferable. Unless, if it is compression ratio that we aim for, then SAMC performs best.

ARM's Thumb

ARM's Thumb (T32) instruction set [ARM],[Goud99] provides a subset of the most commonly used 32-bit ARM instructions which have been compressed into 16-bit wide opcodes. On execution, these 16-bit instructions are decompressed transparently to full 32-bit ARM instructions in real time without performance loss. This way it can offer great code-density for minimal system memory size and cost by having 32-bit performance from an 8 or 16-bit memory.

Thumb-2 technology, as introduced in 2003, made Thumb a mixed (32- and 16-bit) length instruction set, and is the instruction set common to all ARMv7 compliant ARM Cortex implementations. It provides enhanced levels of performance, energy efficiency, and code density as compared to the first Thumb for a wide range of embedded applications. Also, the technology is backwards compatible with earlier ARM and Thumb instruction sets.

There has also been work on improving the Thumb ISA[Xu04], for further size reduction and timing performance.

MIPS16

MIPS16[Kiss97] is an architecture extension that was introduced to address the code density and bandwidth issues of MIPS RISC designs. It was classified as an "architecture extension", because even though it was the standard mechanism for code compression in next MIPS RISC CPUs, support for it was not mandatory for all future implementations. MIPS16 was designed to be fully compatible with existing 32-bit and 64-bit MIPS architectures. MIPS16 instructions can be mapped and executed on a standard MIPS architecture, because they can be translated into 64-bit MIPS-III instructions real-time using simple hardware.

In order to achieve the desired compression, the MIPS16 had to cut down on the MIPS instruction encoding, in all parts. To accomplice that, statistical data from MIPS applications were gathered, to exploit the frequency of the instructions used and also the number of registers. The results showed which instructions were the most important and also that the compiler-generated code rarely used more than 8 registers. So the size of the opcode and operand parts were reduced, thus decreasing instruction flexibility and the number of accessible registers from 32 to 8, but the greatest gain due to reducing the size of the immediate values from 16 bits to 5 (fig 2.11).

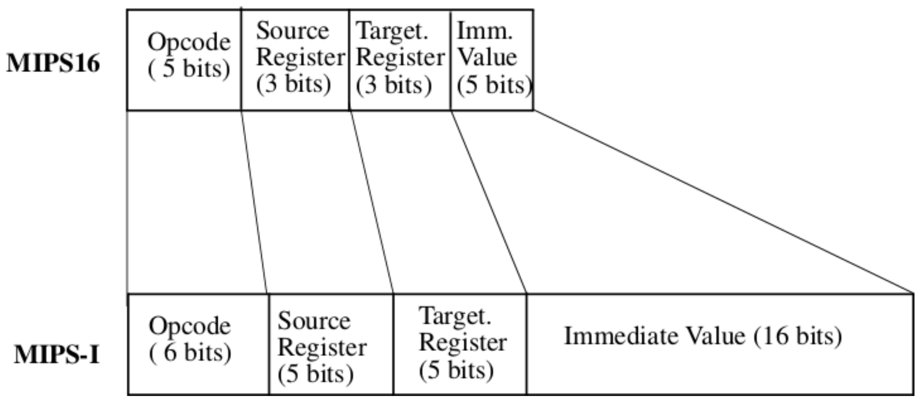


Figure 2.11: Mapping of MIPS16 compressed instructions [Kiss97]

To overcome the shortcomings caused by the compressed instruction set, several specialized mechanisms were developed, mainly aimed at PC and SP relative addressing and extra load stores for larger than 5-bit immediate values.

The results from using the MIPS16 instruction set show that even though more instructions are generated for the same operation and the instructions themselves are now less flexible and expressive, the net code generated by the compiler for a range of desktop and embedded applications is decreased by an average of 40%. Furthermore, the higher code density and the smaller instruction memory contribute to a better hit ratio for the instruction cache and reduced off-chip bandwidth requirements, that more than make up for the slight increase in the absolute number of instructions.

2.5 Architecture Description Languages

Architecture Description Languages (ADLs) [Mish08] are used for designing both hardware and software architectures. Hardware ADLs capture the structure (hardware components, interconnections) as well as the behavior as it is defined by the instruction set architecture of a processor. Software ADLs are used to represent and analyze software architectures. For the purposes of this thesis, wherever ADLs are mentioned, that refers to hardware ADLs.

ADLs have been used for many years now as a successful way to describe the specifications and functions of a processor. The ADL description is used for the generation of several executable models, i.e. the compiler, the hardware implementation and the simulator. Combining these models enables the designer to automate tasks like compilation, simulation, synthesis, test generation and validation. This way the overall time needed for the design is significantly decreased and the quality of the final output improved.

There are several different kinds of ADLs, each one developed for specific purposes. They are sorted into three categories, depending on the nature of the information they describe. Those are: (a) *Structural ADLs* that capture the structure in terms of architectural components and their connectivity and are mostly synthesis and validation oriented, (b) *Behavioral ADLs* that capture the behavior of the instruction-set that belongs to a processor architecture and are compilation and simulation oriented, and finally (c) *Mixed ADLs* that capture both the structure and the behavior of the architecture and support all objective orientations (compilation, simulation, synthesis and validation). Notable examples of ADLs include MIMOLA (structural), ISDL (behavioral), nML (mixed) and Lisa (mixed).

2.6 Design methodologies for ASIP

The definition of *ASIP* as Application Specific Instruction-Set processor was used since the late 1980's [Wolfe88], also mentioned as "Application-Specific Integrated

Processor" in other books. An ASIP is often a Silicon Intellectual Property (SIP), and many SoC solutions use ASIP IP. The main difference between a general purpose processor and an ASIP is the target application domain that defines an ASIP. General purpose processors have to be adequately effective for virtually all possible applications and that is the reason why they cannot by definition be optimal for all, unlike ASIPs that are designed for a specific application domain. The term *application domain* denotes a set of applications that serve the same purpose in similar ways or similar purposes altogether. They usually have the same properties and characteristics and usually benefit equally from certain optimizations. Video decoding, digital radio baseband, or bio-imaging are characteristic examples of application domains. So, an ASIP is fine tuned to be optimal for an application domain, aiming for a higher mix of flexibility, low power consumption and cost and performance than general purpose processors can provide.

ASIPs use a sophisticated hardware and software co-design as shown in figure 2.12, that combines the instruction-set and the hardware components of the processor architecture with the compiler and the application code required for that particular processor. Before each step of the tool flow, the hardware and software flows interact to assure compatibility with each other and optimal implementation.

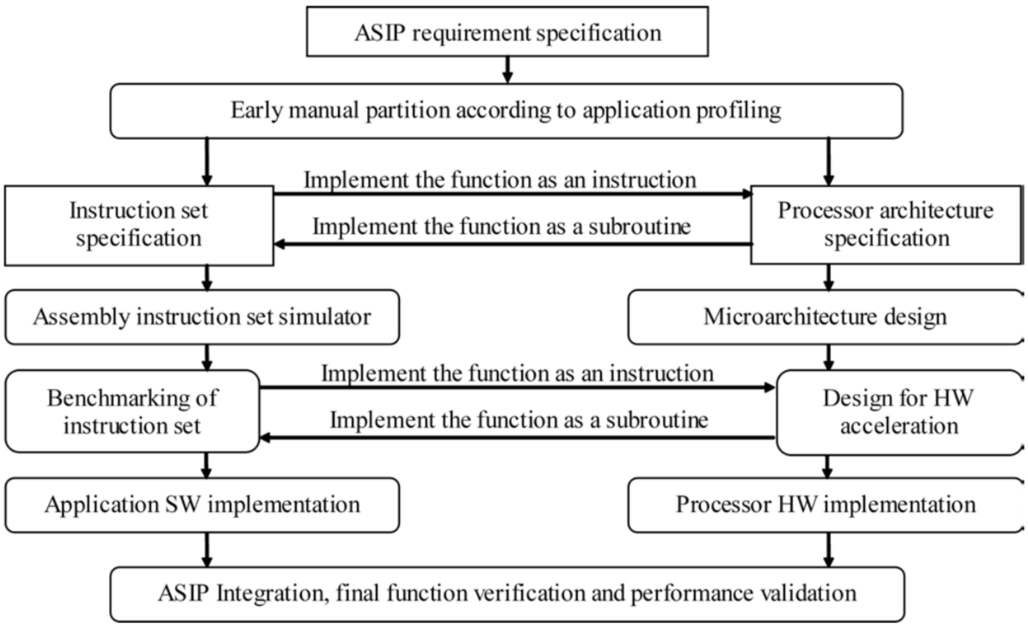

ASIP requirement specification
Early manual partition according to application profiling
Implement the function as an instruction
Instruction set specification
Implement the function as a subroutine
Processor architecture specification
Assembly instruction set simulator
Microarchitecture design
Implement the function as an instruction
Benchmarking of instruction set
Implement the function as a subroutine
Design for HW acceleration
Application SW implementation
Processor HW implementation
ASIP Integration, final function verification and performance validation

Figure 2.12: ASIP hardware/software co-design flow [Liu08]

One of the very first Application Specific Instruction-Set Processor (ASIP) design methodologies that came up was *"Cathedral II"* from IMEC [DeMan86], [Goos87], [Catth88], [DeMan88]. The innovation of "Cathedral II" was the development of an application specific silicon compiler for highly complex DSP algorithms provided that there are defined limitation on the target silicon architecture for a restricted application area. Following a "meet in the middle" design method, Cathedral II enabled design from a high-level behavioural language called SILAGE (oriented for DSP), then rule-based synthesis for the target architecture and generation of the microcode for the controllers and interprocessor communication through heuristic scheduling.

2.6.1 Target IP Designer

The *Target tool flow "IP Designer"* is one of the most advanced products for ASIP design. Target Compilers Technology [Target] provides a fully developed tool flow, equipped with the nML grammar, a mixed ADL with support for compilation, verification, application simulation and HDL generation tools.

More about the Target tool flow will be analysed extensively on a later chapter.

2.6.2 Tensilica's Xtensa

Tensilica's Xtensa [Tensilica] makes use of an ASIP-like methodology named DPU. *Dataplane processors (DPUs)* are designed to provide programmability in the performance-intensive dataplane of the SOC design. They are a combination of a DSP and a CPU, but can be customized for maximum efficiency for the target application. Wide datapaths or instructions can be build into a custom DPU.

Tensilica provides SOC designers with everything needed to quickly design small, low power and high-speed dataplane processors that exactly match the required application. By using Tensilica's Xtensa dataplane processing units (DPUs), design teams can reduce the development and verification time required by hand-coding RTL blocks in Verilog or VHDL. As these DPUs provide programmability into the dataplane, changes can be made in firmware after silicon production that extend the life of the product as standards develop and market needs change.

All Xtensa customizable processors have two essential features, configurability and extensibility. This way, designers are offered a menu of checkbox and drop-down menu options so they can pick just the features they need - including multiple pre-verified DSP engines. Also they can add their own instructions, registers, register files, and much more using the Tensilica Instruction Extension (TIE) methodology. The designer only has to specify the functional behavior of the new data path elements in the TIE language (Verilog-like) and then the RTL and whole tool chain is automatically generated.

2.6.3 LISA and Synopsys Processor Designer

LISA was initially developed by LISATek , afterwards owned by CoWare and finally now integrated in Synopsys Processor Designer [Synopsys]. The

Synopsys Processor Designer is an automated, application-specific embedded processor design and optimization environment that can decrease the time spent on a hardware processor design and the creation of application-specific software development tools. It is highly automated to enable improved architectural exploration and application-specific processor development, as well as consistency checking and individual tool verification. It can be used for the development of a wide range of processor architectures, including SIMD and VLIW and can also support DSP and RISC features.

At the “heart” of Processor Designer lies an ADL named LISA 2.0, a Language of ISAs. LISA 2.0 can use ANSI-C which makes it easy to import existing C/C++ based models and functionality. It also includes an Instruction Set Simulator (ISS), and a complete software development suite with assembler, linker, archiver, C-compiler and synthesizable RTL code. Furthermore, it provides profiling capabilities in the debugger, rapid analysis and exploration of an ISA. The instruction set design, the processor’s micro-architecture and the memory subsystems can be independently optimized.

2.7 ASIP Case studies

There are several case studies published with ASIP being employed to face similar problems. Some of the most interesting ones due to their innovative ides or impressive results are analyzed here.

In [Morg07] a code compression technique is used to make a more compact instruction. During software analysis, the opcodes dispatched to individual functional units of a VLIW processor are measured (fig. 2.13. Using that information, a dictionary-like encoding scheme is created at a more fine-grained level than other approaches (e.g. [Piguet01]). To make sure that the lookup table will be reasonable, instead of encoding all possible opcodes for each functional unit, only the frequently used opcodes identified from the profile-based analysis remain in the LUT.

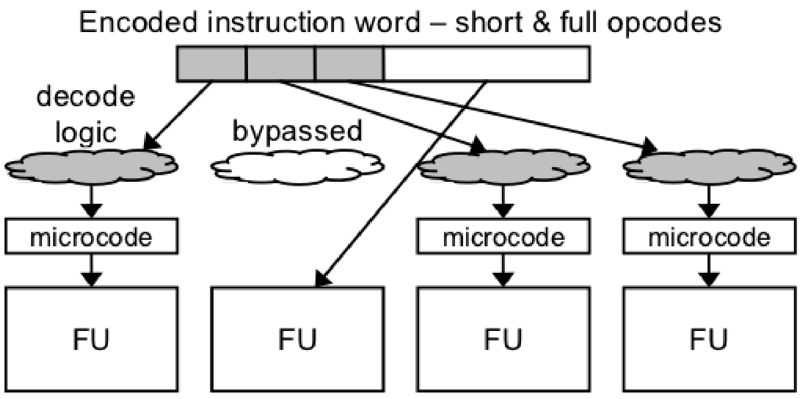


Figure 2.13: Different design styles target different design metrics [Morg07]

Each short opcode within the instruction word is split into two sections. The selector "address" indicating the functional unit and the encoded instruction. A Huffman-type encoding is used to allocate variable-width addresses in priority order of FU usage. This way overburdened FUs have more active opcode bits and need less address bits. Each FU has its own unique LUT decode logic to decode its short opcode into microcode, as well as an escape code instruction indicating that a full opcode should be fetched from the instruction word. The short opcode usually ends up with a width of 8 to 11 bits for optimal results. If more then it would require large and inefficient decode logic and if less it would be too restrictive on the available short opcodes. The algorithm employed dynamically estimates if the cost of increasing the FU short opcode width by a single bit (thus doubling the number of available short opcodes) is worth the extra cost.

To make this decision the algorithm compares the benefit of energy gain from the increase in opcodes against the cost of the extra decode logic. The results from this implementation show considerable cache area decrease in almost all tests, 18% decrease in the area and a slight increase in performance (due to the decrease in clock cycles by 8%). In terms of energy the total energy has dropped by 20

A very interesting framework for optimizing mainly power-wise the instruction encoding of an already existing ASIP processor is provided in [Chat07], with [Zhang08] taking the same project one step further by adding a more automated algorithmic approach. The focus in this implementation is on how to reduce the power consumption by emphasizing on both the self and coupling capacitance of the bus lines, separating the different possible bit transitions and calculating as precisely as possible the cost of these transitions.

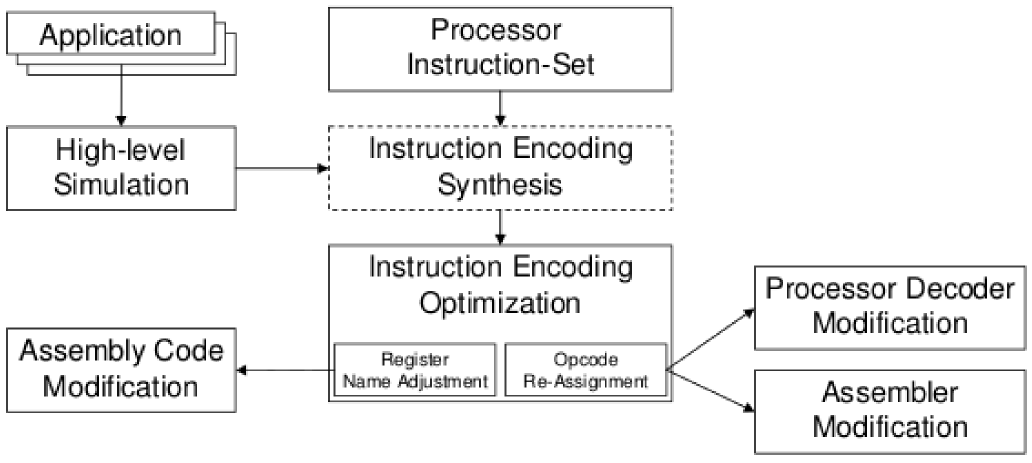

Application
Processor Instruction-Set
High-level Simulation
Instruction Encoding Synthesis
Instruction Encoding Optimization
Register Name Adjustment
Opcode Re-Assignment
Assembly Code Modification
Processor Decoder Modification
Assembler Modification

Figure 2.14: Overall encoding synthesis flow [Zhang08]

That power model is then used along with the ADL grammar file describing the instruction-set and with an assembly program in a series of algorithmic optimizations (figure 2.14). First the opcode itself is changed to minimize power consumption with regard to the applications provided by performing the *Opcode Re-Assignment (ORA)*. The ORA technique breaks down the instruction set, tracks down the dependencies between the various instructions, simulates the assembly program and maps the toggling and coupling results to its corresponding instructions. The directed graphs created this way are then grouped into column graphs according to each dependent instruction to ensure unique encoding. Also a hash table is generated from the coupling information between the nodes. To produce the updated grammar file, first an initial coding is assigned through gray coding and then a heuristic optimization method is applied attempting to find an encoding that is most power efficient.

With the updated grammar file the *Register Name Adjustment (RNA)* is applied on the application code, rearranging it so that is compatible with the new grammar and also using the statistics from information extraction to perform a heuristic approach similar to ORA so that the register file usage is power-efficient as well.

But what happens if instead of a single program there are are more programs, commonly referred to as an application domain. In that case, as shown on figure 2.15 the columns information is extracted for each different assembly program and a single set of column graphs and a hash table are created. Those are taken into account and analyzed with a unique updated grammar file generated through the ORA technique. That new grammar information is used as the common grammar information for all the programs. Using that, the RNA technique is performed for each program *individually*, and out of each program a register graph for toggling information and a hash table for coupling information are created for re-assigning the registers.

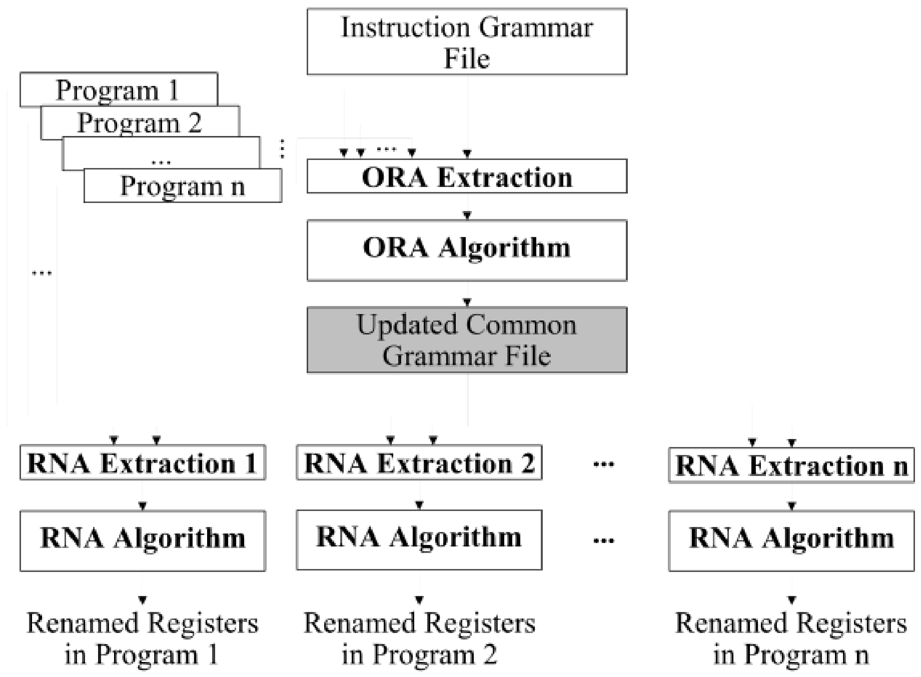


Figure 2.15: Multiple assembly programs optimization flow [Zhang08]

Chapter 3

Development Framework

In this chapter the basic development framework that was used for this thesis is provided, along with the particular ASIP methodology employed to design different ASIP processor implementations.

The Tools that were used for the purpose of this thesis were Target's retargetable tool-suite "IP Designer" for designing, compiling, programming, simulation and verification of ASIP cores and their applications.

3.1 nML Grammar

The whole tool flow has at its heart the nML grammar (figure 3.1). *nML* is a hierarchical and highly structured ADL. It models a processor in a concise way for a retargetable processor design and software development tool suite. It has been designed to contain the right amount of hardware knowledge that is required by the Target tools for high quality results.

A unique feature of Target's Chess/Checkers tool suite is its architectural retargetability, based on the nML processor description language. nML is a high-level language that captures a programmer's model of the target processor. This is the abstraction level commonly found in a programmer's manual of a processor. Using nML, an architecture designer can quickly define the instruction-set architecture of a processor or make any changes without wasting much time. After reading the nML description, the different tools are automatically targeted to the specific architecture.

It should be made clear that the term *Retargetable* is used to describe that the tools (including the C compiler) are targeted towards the architecture so described in nML. There is no restriction to the type of ISA that can be modelled: RISC, CISM, SIMD, VLIW, integer and floating-point architectures are supported. All tools take the nML model into account. and the retargeting process is very fast, since a few seconds are more than enough to retarget the compiler, simulator and all other

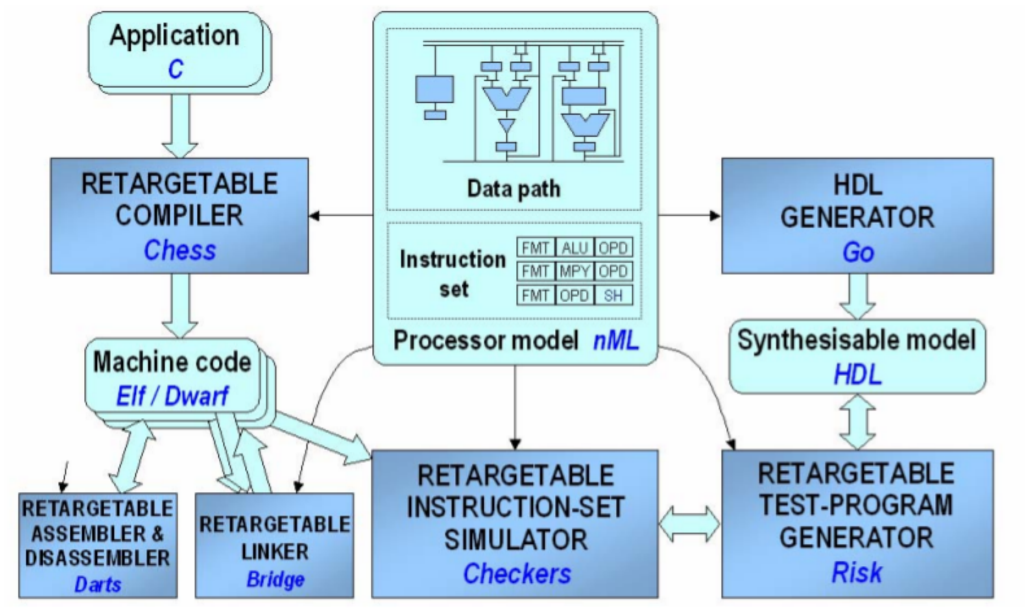


Figure 3.1: Outline of Target's Chess/Checkers tool suite flow

tools.

Below are listed the basic definitions of the nML grammar[Mish08]:

3.1.1 nML Structural Skeleton

- Memory:
  `mem DM[0..1023,1]<num,addr>;`
- Register:
  `reg X[4]<num,b2u>;`
- Constant:
  `cst c2u<uint2>;`
- Enumeration:
  `enum alu{add, sub, and, or};`
- Transitory:
  `trn A<num>;`
- Pipe Register:
  `pipe F<acc>;`
- Functional Unit:
  `fu alu;`

3.1.2 nML Rule Definition

Example of an ALU definition:

```
opn alu( op: alu_op, t: rt, r: rr, s: rs ) {
  action {
    stage E1:
    alur = r;
    alus = s;
    switch(op) {
      case add: alut = add(alur, alus) @alu
      case sub: alut = sub(alur, alus) @alu
      case and: alut = and(alur, alus) @alu
      case or: alut = or(alur, alus) @alu
    }
    t = alut;
  }
  syntax : op " " t "," r "," s;
  image : op::t::r::s;
}
```

We have two kind of rules. *OR* rules (symbolized by "|") alternatives for an instruction part. These alternatives are mutually exclusive, meaning that only one of them can be executed at a time. *AND* rules (symbolized by "::") are the rules that describe the composition of instruction parts. The composing instruction parts are orthogonal, meaning that the concatenation of any legal derivation for every instruction part forms a legal derivation for the AND rule itself.

In the code above, we should pay attention to three attributes:

- The *action* attribute describes which register transfer actions are performed by an instruction or instruction part. Each AND rule must have one action attribute.

- The *syntax* attribute specifies the assembler syntax (mnemonics) for the corresponding instruction (part). It must only be present if the intention is to derive an assembler or disassembler tool from the nML description. An AND rule may have multiple syntax attributes if needed.

- The *image* attribute defines the binary encoding for the corresponding instruction or instruction part. In some cases, multiple image attributes may be needed.

It should be noted that while an OR rule does not need any explicitly defined attributes, it implicitly passes attributes between its left and its right-hand sides.

3.1.3 Primitives definition and generation language

To keep in line with the need of the ASIP hardware/software co-design, each of the operations are matched to a primitive function and subsequently a function in the Primitives Definition and Generation (PDG) language (.p) file. These functions provide the functional description of the instruction in C language, so that the software tools can create the compiler for that particular processor, which in turn compiles the application code meant for this processor in an assembly that it can process and execute.

3.2 Target tool flow

The tools as shown in figure 3.1 include the following:

3.2.1 Chess

Chess is a *retargetable C compiler* that translates C source code into machine code for the target processor. It uses graph-based modelling and optimization techniques to generate optimized code for specialized architectures exhibiting peculiarities such as complex instruction pipelines, heterogeneous register structures, specialized functional units and in-level parallelism.

The compiler also includes a retargetable assembler and disassembler called *Darts* and a retargetable linker called *Bridge*.

3.2.2 Checkers

A *retargetable instruction-set simulator (ISS) generator* that produces a cycle-accurate or bit-accurate ISS for the target processor based on the nML description. The ISS can be run in stand-alone mode or can be embedded in a co-simulation environment through an application programming interface (API). This allows the designer to simulate the C code generated by Chess with the instruction-set architecture. Checkers also includes a graphical debugger that can connect both to the ISS and to the available processor hardware for on-chip debugging, through the JTAG interface. Profiling and instruction trace are also supported.

3.2.3 Go

A *hardware description language (HDL) generator* that produces a synthesisable RTL (register-transfer-level) HDL model of the target processor core. Through APIs, users can plug in their own HDL implementations of functional units and of the memory architecture.

3.2.4 Risk

A *retargetable test-program generator* that can generate assembly-level test-programs for the target processor with a high fault coverage. These test programs can then be executed both in the ISS and in the HDL model of the processor, to check for the consistency of both models.

3.3 Instuction set encoding

In an embedded processor, the way an instruction set is encoded affects directly the way the instructions themselves are encoded in binary and thus executed by the processor. Therefore this affects likewise not only the size of the executed code but the way the instruction decoder itself is build so that it can efficiently decode these binary instructions, with fast access to the operation and the operands. The first part that refers to the operation is usually referred to as the *opcode*. The opcode also includes the information about the addressing modes of the different operations.

The encoding of the instruction set in a processor has a very strong influence on the rest of the design. It also defines two important parameters in a processor: The size of the program memory and the available flexibility. The size of the program memory is a defining parameter is the overall energy consumption. So for a smaller instruction width, a smaller program memory is needed, but a part of the flexibility is sacrificed. Finding a suitable trade-off between flexibility and code size for an optimal instruction encoding is quite challenging, and also depends greatly on the application domain.

An option that is especially popular with VLIW processors is the compression of the instruction set. However, in that case extra area and power is needed to uncompress and decode the instructions.

When developing an ASIP, some developers tend to overlook the encoding of the instruction set or simply try to make the instruction set as compact as possible to reduce the instruction width.

In the Target tool suite, the instruction encoding is defined in the *opn rules* of the nML Grammar (sec: 3.1). All of the available encoding schemes can be applied in the nML, so that they can improve the resulting processor design. More about that on a later chapter.

Chapter 4

Development of a SIMD ASIP

In order to validate the motivation of this thesis a SIMD processor was developed, based on an example processor provided by Target, and then compared to a scalar processor and a soft-SIMD one. The example processor of target is a small basic SIMD vector processor which exemplifies the ability of the Target tools to incorporate SIMD instructions in an ASIP and is typically taken as a starting point for vector processor design. It is also a good starting point for the main implementation that follows in the next chapter and to explore the percentage of energy consumed by each component. The evaluated Soft-SIMD processor is part of a Master Thesis[Dak11] that was conducted in IMEC/Holst Centre very recently and is used to get an understanding of the differences in performance and power consumption of SIMD and Soft-SIMD implementations.

4.1 Hardware SIMD and Software SIMD

Hardware SIMD (Single Instruction Multiple Data), also called hard-SIMD is a vector architecture that supports operation on several data in parallel or operations on several narrower data types at the same time by treating a single register as if it contains multiple data words. For example, four 16-bit data additions could be executed on a 64-bit ALU in a single cycle in parallel, provided the carries of the separate additions are isolated. SIMD processors use special hardware in their data-path for computations on a certain combination of sub-words of the same lengths. A common example of this would be treating a 32-bit (word) register as containing two 16-bit (half-word) data or four 8-bit (byte) data, and then being able to perform an operation on each sub-word. This is what differentiates SIMD processors from normal vector processors that only support a single parallel execution mode (e.g. 10 x 32 bits) [Catth10]. This offers new parallelization options allowing multiple smaller operations to be performed on each of the smaller data type.

SIMD operations may require additional hardware but they offer great performance and can accelerate algorithm operations and applications with parallel operations or loops that are repeated a lot (e.g. vector operations and digital filters). Naturally, SIMD processors are ineffective in executing serial code or code with bad memory locality. Therefore, to enhance the SIMD capabilities of a processor, extra instructions are needed for packing (or compressing) and unpacking (or uncompressing) SIMD data in and out of registers (fig. 4.1). The overhead of this packing and unpacking instructions depends on how many times the data are used and where they are stored. Even though SIMD operations increase power consumption, they greatly decrease the number of cycles needed to perform a task, which depending on the application can sometimes lead to smaller energy consumption for the same task.

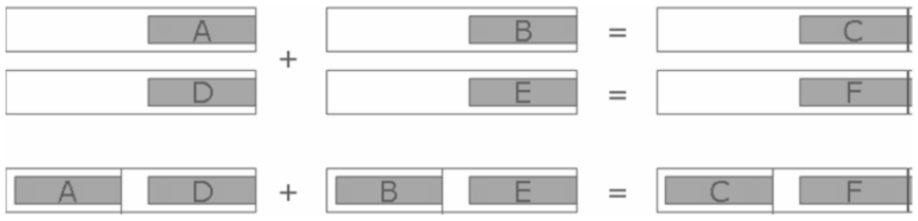


Figure 4.1: SIMD data packing

The reason for the classification between hardware and software SIMD is because hardware SIMD is performed in hardware level and requires extra hardware, along with all the extra power and area that requires. A *Software SIMD (Soft-SIMD) processor*, unlike a hardware SIMD, attempts to do the same procedure, but it is instead emulated in software and the application code needs to be prepared by the compiler. This method is fruitful only if the application code can be parallelized and the processor data width is big enough to support a sufficient number of sub-words. For example, applying soft-SIMD to enable the execution of two 8-bit sub-words in 16-bit ALU is not as beneficial as enabling the execution of eight 8-bit sub-words in a 64-bit ALU. That is because the overhead of the soft-SIMD implementation is roughly the same, independently of the number of sub-words executed in parallel.

Unlike Hardware SIMD, Soft-SIMD implementations do not require extra hardware and for that reason they can be safely assumed to consume less power. However, to make up for that they have to support extra operations. Evidently, there is a trade-off that has to be met, and sometimes the overhead of the extra operations can surpass the one of the extra hardware. Soft-SIMD has the advantage of being more adaptive, allowing the execution of a variety of combinations of different types of data sizes. This increases the potential and the benefits of a Soft-SIMD but also increases the complexity of the design and the expected gain.

4.2 Basic features of the VBase processor

The *VBase* example core demonstrates the modelling of SIMD instructions. As mentioned earlier, SIMD stands for single instruction stream, multiple data streams. SIMD instructions are also known as vector instructions. In SIMD parallel processing, a vector of data is stored in a vector register. A SIMD instruction processes the elements of the vector simultaneously. All elements are processed in a identical way, as specified by the single instruction. By processing whole vectors of data at once, SIMD provides a fast and efficient way to manipulate large amounts of data.

A drawback of SIMD is that it requires the computational kernels of the applications written in a way to efficiently use the SIMD instructions. However in the Target tool flow, by using the Chess compiler, this can still be done at C source code level thanks to the availability of an extended type system and intrinsics. The vector data types can be used in the C code along with vector intrinsic functions and operators defined for the vector types.

The VBase core contains:

- 16-bit instruction word
- 16-bit data word
- Separate instruction and data memory. The vector memory is mapped and aligned on the data memory using the *alias* nML grammar option.
- A 128 bit vector unit.
- A 128 bit vector register file.
- A 128 bit vector memory data port.

VBase supports vectors of 16 elements of one byte each and vectors of 8 elements of one word each.

4.3 Additions and modifications

For the purposes of this case study, several modifications were made to the VBase processor:

- The number of elements processed was changed to 4 elements of one word each (SIMD slots), so the processor is now able to support a vector of 4 elements of 16 bits each.
- New instructions were added that can control a multiply-accumulate (MAC) functional unit, divided in two stages (fig. 4.3).

- Also added instruction for initializing or resetting the accumulator, as well as instructions for extracting the 16-bit MS or 16-bit LS part of an addition before that is forwarded to the shifter (and then shortened).
- Also, extra units were used to handle the accumulator and the rounding.
- New data types were introduced to handle the inputs, outputs and intermediate data of the MAC. Mainly the vector types, accumulator type, and a slightly adjusted instruction word type.
- Changed the memory management to allow for a separate Vector Memory, instead of mapping and aligning it to the existing Data Memory.

4.4 Gauss loop filtering

To examine and check the performance of an SIMD implementation the MAC unit analysed before was tailored to execute the filtering of an image in a bio-imaging application. The same was also used in the Soft-SIMD design.

The focus of the application is on the *critical Gauss loop* where the majority of cycles is spent. The loop is what has the maximum number of constant multiplications in the whole application. That loop applies the Gauss filter to a frame in a detection algorithm. In the original code, the Gauss filter is applied through a 3 x 3 matrix that hold the Gauss coefficients (fig. 4.2).

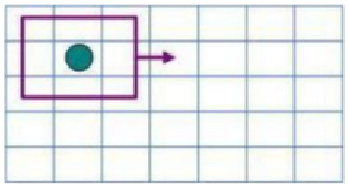


Figure 4.2: Application of Gauss filter through a coefficient square matrix [Psy10]

The result of the application of the Gauss filter for one pixel is calculated in every iteration using the eight neighbouring pixels. Each neighbouring pixel and the one in the centre are multiplied with the corresponding coefficient. The results are summed up and the final value replaces the one in the central pixel. The aim of this application is to reduce the noise in an image to enhance the detection algorithm of the application which uses an ellipse to detect the location and posture of the monitored object.

This application is handled in a bit different way by each implementation, and so the specifics for its execution will be analyzed independently in a later section.

4.5 Multiply-accumulate unit

At the heart of any DSP lies the *multiply-accumulate unit*, commonly referred to as *MAC*. By multiply-accumulate we denote the sum of multiplications used in digital filters, correlations and Fast Fourier Transforms. Ideally the MAC operation should be executed in a single cycle (so that the *CPI* is 1) inside a pipelined architecture. The accumulator should be big enough to accommodate the expected growth in size of the result. The result of a multiplication of two 16-bit integers would be 32 bits and adding another 32-bit integer would normally result to a 33-bit integer. But because of the range of data that is processed there is not need for the extra overflow bit (more about that later). Otherwise there would be a need for guard bits in case of arithmetic overflow. Usually for 16-bit data the accumulators are 40 bits, with 8 guard bits that help save overflow information.

The schematic of the MAC unit designed for the purposes of this thesis can be seen in figure 4.3. The MAC unit supports a reset and initialization function for the 32-bit accumulator. Using the two input ports of the accumulator the 16-bit data are inserted and the result of their multiplication is added to the current value of the accumulator. The result of the addition is then stored as the new value of the accumulator. Depending on the instruction being executed there is also the option of storing the result of the ALU in the vector register file, provided it is first saturated in a safe way back to the size of 16 bits.

It should be noted that while the two inputs of the multiplier look alike, the first one comes from a part of the vector register read, so its a 16-bit part of 64-bit vector register data while the second is the coefficient and is the same for all four of them.

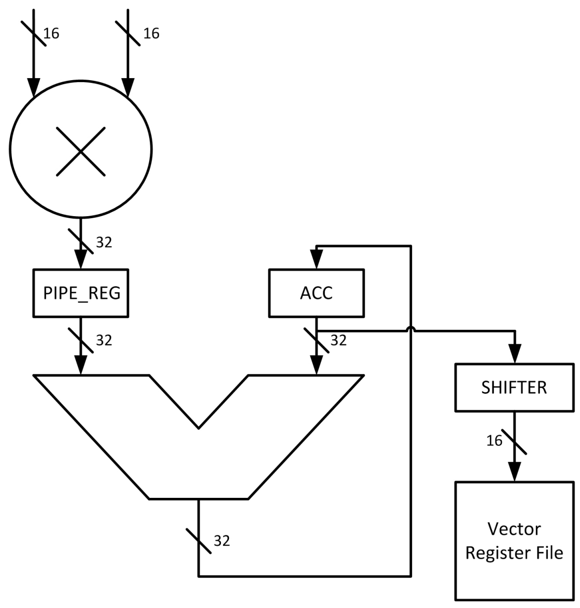


Figure 4.3: The basic MAC unit introduced

It should be noted that as mentioned before this is an SIMD processor, therefore there are *four instances of this MAC unit* in the design, giving the capability to process 4 x 16bit = 64 bit of data.

On each iteration the following procedure is followed for the application of the filter on 4 pixels at the same time:

```
reset acc
acc = previous_pixel * coeff1
acc = current_pixel * coeff2 + acc
acc = next_pixel * coeff1 + acc
new_current_pixel = shift(acc)
```

4.5.1 Shifter and overflow prevention

The shifter in the design scales the results of the ALU to avoid overflows. DSP architectures commonly use *saturating arithmetic*, so if the result is too large to be stored and represented then it is set to the largest representable number, which also depends on whether it is signed or unsigned and if it is signed then the sign of the number.

To calculate which are the most important bits in the result it is first needed to calculate the maximum number that it might be. Its helpful to know that the numbers are unsigned. According to the algorithm followed above the following function comes up: $result = (a + c) * coeff1 + b * coeff2$, where teach of the a,b,c is a pixel with a size of 7 bits and the two coefficients have a sign of 10 bits, then the maximum number can fit in 19 bits. The maximum size of the first multiplication is 18 bits, because (7+7)*10bits = 8*10 = 18 bits) and the one of the second is (7*10)bits = 17 bits, then 18 bits added with 17 bits need a maximum size of 19 bits. That means that the rest can be skipped and shift the number 13 bits (since it is unsigned) and keep the MS bits that fit.

4.5.2 Wrapper and Testbench

The wrapper used used in the testing, as shown in figure 4.4, is connecting the vbase processor with the three different memories used. This way the processor can access at any time all of the three memories.

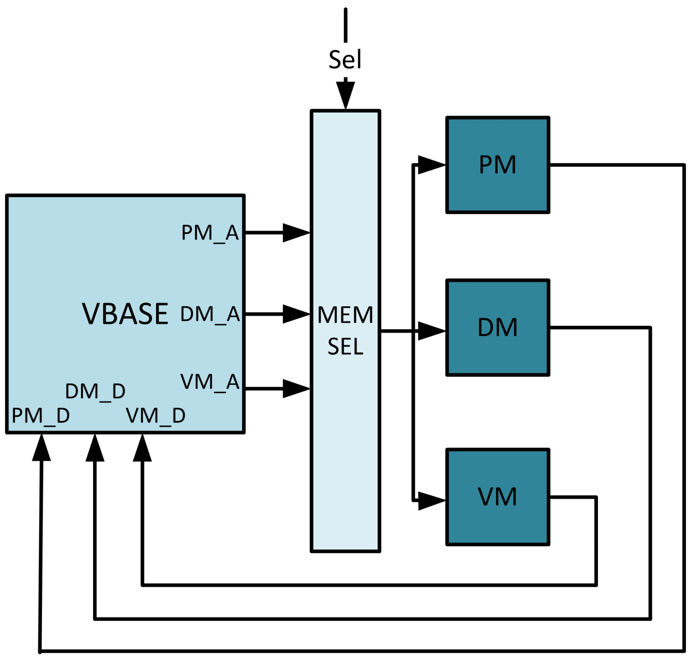


Figure 4.4: The wrapper outline

4.5.3 Additional instructions added

MAC instructions, just as the name implies perform the multiply and accumulate operations in the dvmac FU. They have to be included in the same opn rule as is shown below, to keep the pipeline intact. In the E1 stage, the multiply is performed and the result is stored in the pipe register, and in the E2 stage the addition is performed using the accumulator and the stored result in the pipeline register.

```
opn vec_dvvs(vreg, reg) // MAC operations
{
    action {
    stage E1:
        dv_pipe = dv_mul_acc(vreg, reg) @dvmac;
    stage E2:
        acc = dv_add_acc(dv_pipe, acc) @dvmac;
}
```

Accumulator instructions use the dvmac FU and either return the MS or LS 16-bit part of the 32-bit accumulator or perform the shift operation as it was explained before to extract the most important data bits and store them in the vector register file. Also there is an instruction that resets the content of the accumulator to ensure it is empty when used and won't pollute the current iteration with previous ones.

```
V[x] = acc_ms(accr) @dvmac;
V[x] = acc_ls(accr) @dvmac;
V[x] = vshift(accr) @dvmac;
acc = dvinit() @dvmac;
```

Packing/Unpacking instructions use the vec FU and allow either the insertion of a 16-bit data from the register file to a specific one of the four spots in the 64-bit SIMD data or the extraction of one of them.

```
V[x] = vec_insert(vector_register, spot, data) @vec;
reg = vec_extract(vector_register, spot) @vec;
```

Naturally, to keep in line with the need of the ASIP hardware/software co-design, each of these operations were matched to a primitive function and subsequently a function in the Primitives Definition and Generation (PDG) language (.p) file.

4.6 Comparison of the hard-SIMD with a soft-SIMD implementations

4.6.1 The Soft-SIMD implementation

There have been various Soft-SIMD implementations for a solution to the critical Gauss loop mentioned above (see sec. 4.4) in a series of theses [Krit09], [Psy10], [Dak11]. The one used for the comparison that follows is the most recent one by S. Dakourou.

In this particular Soft-SIMD implementation, the designer is taking advantage of the adaptive features of Soft-SIMD by using three different subword combinations to execute the Gauss filter application. 6 x 8 bits, 4 x 12 bits and 3 x 16 bits, as shown in figure 4.5, always keeping a total of 48 bits. It is an architecture aimed for loop dominated domains, exhibiting sufficient data level parallelism, with signals of multiple word-lengths and a relatively small number of multiplications.

The outline of the architecture, as illustrated in fig. 4.6, is using a GSAS FU (Generic Shift Add Shuffle Function Unit) architecture, consisting of a shifter,

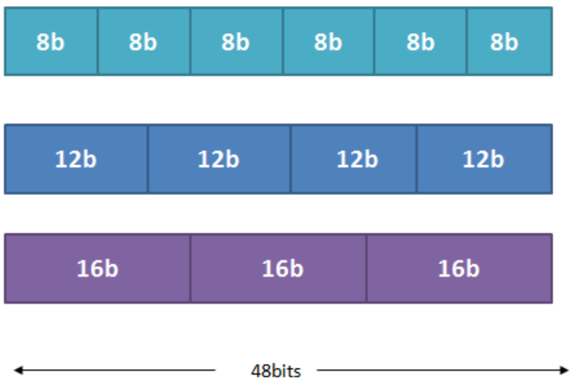


Figure 4.5: Soft-SIMD sub-words [Dak11]

an adder and a shuffler. The *shuffler* is used to handle the required masking operations needed for the Soft-SIMD. The shuffler is also in charge of choosing an intermediate subword size instead of using by default the worst-case subword size. Special repacking operations are applied whenever a change in subword size is decided. To make the critical Gauss loop Soft-SIMD compatible it is split into two loops. That reduces the total number of multiplications and also allows efficient scheduling in the data-path. The shuffler is the component that maintains the functional correctness by handling the subword manipulations.

Instead of costly hardware multipliers this implementation includes a *shift-add functional unit*, which uses a number of shift and add operations to implement the constant multiplications required by the Gauss filtering within the Soft-SIMD concept.

The architecture is also making use of a novel asymmetric register file organization called *Very Wide Register File (VWR)* or foreground memory organization. It uses asymmetric interfaces: a wide interface with the memory and a narrow one to the data-path, so it only has a single port per cell and is more power efficient.

There are *special operations* added to handle the packing, repacking and unpacking of the data, and also guard intervals that guarantee that each subword does not overwrite nearby subwords and avoid data pollution.

The *instruction-set* used has an instruction word of 80 bits and is divided as follows:

- Issue slot 1: Target Base core FU, 16 control its
- Issue slot 2: Multiplexers network and vector shift-add FU, 37 control bits
- Issue slot 3: Shuffler FU, 11 control bits
- Issue slot 4: Interaction DM & RF, 16 control bits

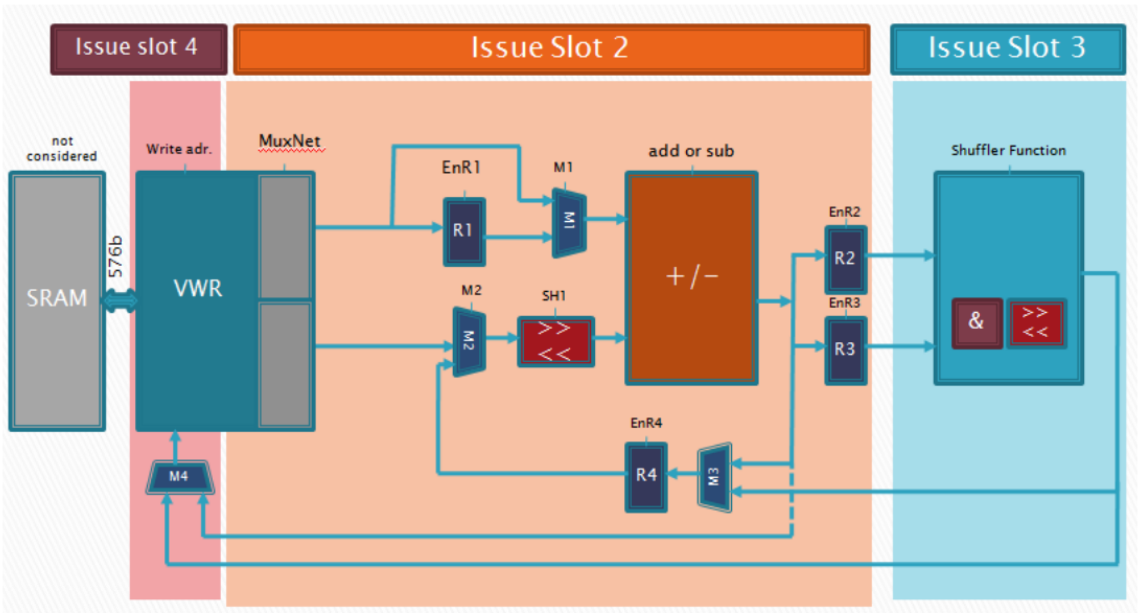


Figure 4.6: Soft-SIMD processor architecture [Dak11]

4.6.2 Comparison and results

The power simulations are performed on a synthesized netlist with a frequency of 100MHz using the 90nm-LP TSMC libraries. It is not as accurate as a "place and route" layout would be, but still it can provide the results needed without going through the trouble and extensive time needed for place and route every time.

In the wrapper, the Soft-SIMD is using a 1024 x 80-bit program memory, a 1024 x 16-bit data memory and 8 x 1024 x 48-bit vector memories. In the Hard-SIMD because of the fact that at the time there were no 64-bit memories available for the vector memory, eight copies of the 1024 x 80-bit memory of the Soft-SIMD are used instead, and the unused part are set to zeros "0". For the rest, the 1024 x 80-bit program memory (which is way larger than the 16-bit that should have been used instead) and the 1024 x 16-bit data memory were kept as they were. For these reasons, the power consumption of the memories should **not** be taken into account, since they are not optimal for each implementation and therefore not comparable.

The results from the power simulations are shown in the following figures. Analyzing the figures in a fair way is vital both to understanding the motivation behind this thesis and also getting an idea of the impact of an architecture on the synthesized design as it is presented in the ASIP tools.

Analyzing the results of the Hard-SIMD implementation

Before looking into any numbers or comparisons and in order to support the motivation it is essential to look into the percentage of power consumption in each component of the Hard-SIMD implementation (figure 4.7). The memories are clearly dominant, but for reasons mentioned before the focus is on the logic and not the memory part of the design. Therefore, by ignoring the memories which are anyway too big for this design and could be misleading, the decoder takes up a 12% of the logic components, which is substantial considering this is a simple design with a much smaller number of instructions and addressing modes compared to up-to-date commercial ASIP processors. The instruction set of this processor is nowhere near as complex as that of a commercial ASIP one.

As for the rest of the components, it is evident that the scalar functional units along with the data memory are used very rarely and most power consumption comes from the vector MAC unit (containing four multipliers for four parallel multiplications as well as the pipeline registers) and the accumulator.

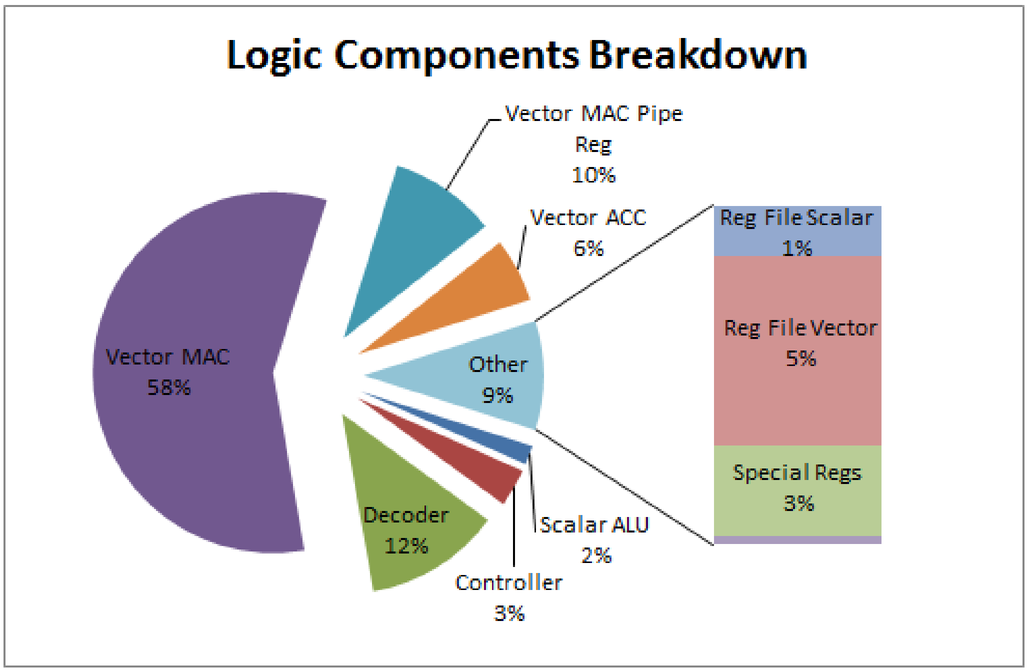


Figure 4.7: Hard-SIMD power consumption percentage in login components

The same conclusions can by reached by looking into figure 4.8, where the switching activity is also accounted for and can it be observed where the dynamic power is spent and which of components are used the most. As would be expected, the dynamic power dissipation takes place in the vector mac unit and the accumulator. As part of the total consumption, the leakage power is only a small fraction, and mostly concentrated in the vector memory.

From the total power consumption, 69% of it is static (internal), 21% is dynamic (switching) and 10% is leakage energy.

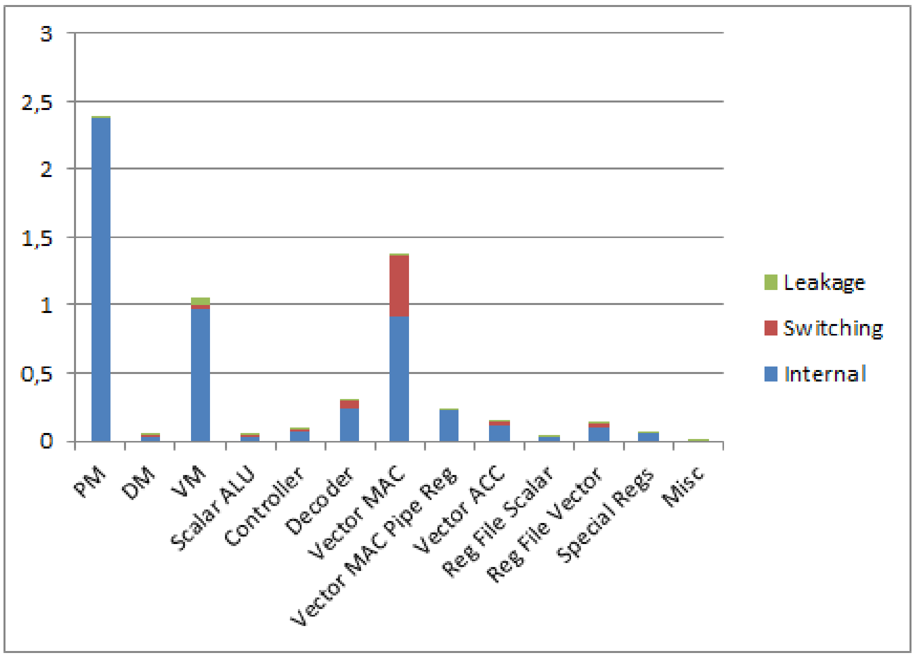


Figure 4.8: Hard-SIMD power type per component

Comparing Target's Base, Hard-SIMD and Soft-SIMD implementations

The results of the simulations of three different processors can be seen in full in table 4.1. First is the sample scalar processor provided by Target, which is *not* the same as the SIMD processor used for the Hard-SIMD implementation and is used as a reference point to scalar processors. Then comes the Hard-SIMD processor developed for this case study and last the Soft-SIMD processor as analyzed before (section 4.6.1). The same results are also illustrated in figure 4.9, with different bars for the different types of power consumption.

The scalar processor cannot possibly match the execution time and cycles needed for this task as the SIMD implementation and needs roughly six times more cycles than the SIMDs. So even though it has a simple small design with an area half or a quarter smaller to the Hard-SIMD and Soft-SIMD respectively, the excessive cycle count also leads to an excessive power consumption almost three times bigger than the SIMDs.

Before jumping to conclusions for the hard and soft-SIMD implementations, the different factors that lead to their power consumption shown on table 4.1 need to be analyzed independently.

The energy consumption and area required for the memories according to the data is virtually the same. As was mentioned before, these two implementations

Processor	Cycles	Area (Cells)	Total Power (W)	Memory Energy (J)	Logic Energy (J)	Total Energy (J)
Target's Base	1972170	7130	2.51E-03	3.80E-05	1.14E-05	4.95E-05
Hard-SIMD	300438	13332	5.87E-03	1.05E-05	0.71E-05	1.76E-05
Soft-SIMD	334032	21625	5.45E-03	1.14E-05	0.68E-05	1.82E-05

Table 4.1: Target's Base cycle count, area, power and energy needed for the same filter application

use almost the same memories, something that proves unfair for the Hard-SIMD implementation which would normally use much smaller memories and is therefore ignored for the rest of results.

Looking into the *area (in cells)* required in each of the two implementations in figure 4.10, it is evident that the Hard-SIMD with its four 16-bit hardware multipliers, along with the pipeline registers and the accumulator in the MAC unit takes up roughly twice the area of the combined area of the vector shift-add unit and the shuffler of the soft-SIMD implementation. For most of the rest of the components the area is relatively the same. The surprising results come from the big difference in the area needed for the decoder and the vector register file. The decoder of the Hard-SIMD is way smaller than the one of the Hard-SIMD, because the former is only 16-bits wide whereas the latter is 80-bits wide. The vector register file of the Hard-SIMD is also much smaller and simpler than the complicated Soft-SIMD vector register file (VRF). For all these reasons, the total area of the Hard-SIMD is 13332 cells as compared to the 21625 cells of the Soft-SIMD. So, the cost (in cells) of the extra hardware multipliers used seems to be smaller than the cost of the bigger decoder and the bigger and more complicated vector register file, not taking into account the program memory used that is 5 times larger in the Soft-SIMD. Of course, the cell count is not totally trustworthy, since modern tools are capable of doing wonders in optimizing area.

Energy consumption, as illustrated in figure 4.11, is the main objective behind all the comparison. It is of course relative to a certain extend on the area as explained before. Looking into the main points as they were also explained in the area analysis before, the four 16-bit hardware multipliers in the Hard-SIMD MAC unit prove to be way more power-hungry than the Soft-SIMD's vector shift-add unit and its shuffler by a factor of 5. But the energy cost of the rest of the component in the Soft-SIMD, especially the decoder, the register file (in the Hard-SIMD it is only seldom used) and the vector register file add up to quite a big sum. This way the total energy consumption of the logic components in the design of the Hard-SIMD is only a bit larger than that of the Soft-SIMD.

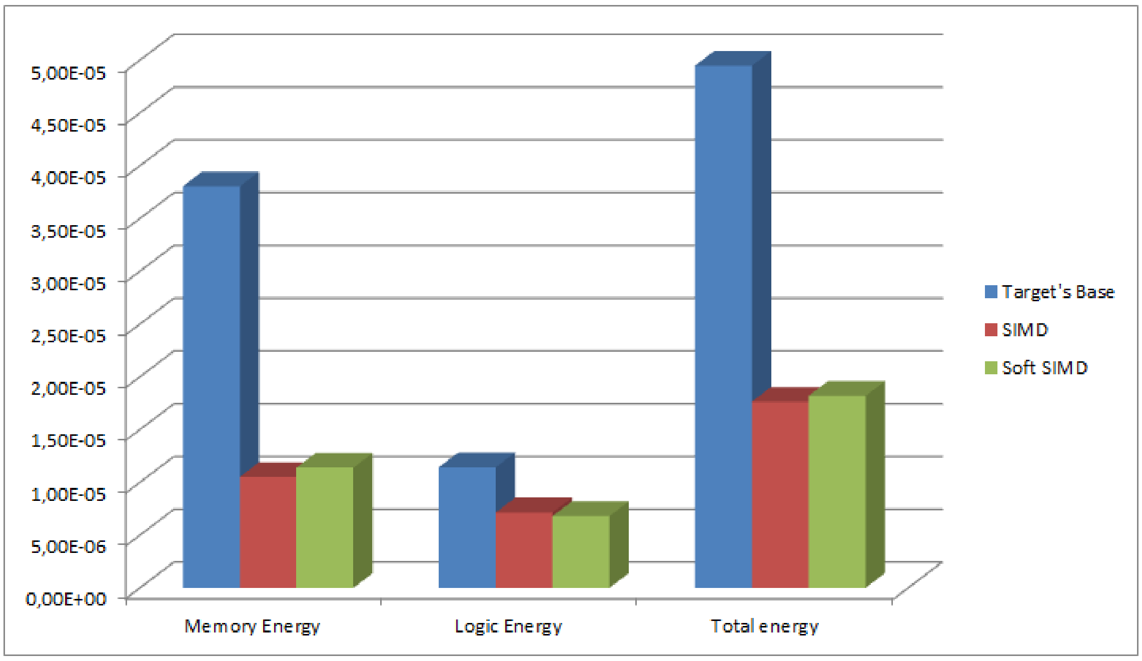


Figure 4.9: Target's Base, Hard-SIMD and Soft-SIMD memory, logic and total energy consumption comparisons

Notes on the comparison from the soft-SIMD development team

Before reaching the conclusions, it is vital to also provide the reasons for these results according to the development team of the soft-SIMD implementation, presented by Francky Catthoor and Stefania Dakourou.

The vector register-file (VRF) unit is mimicking the functional behaviour of the VWR (Very wide register) that should be used in the SoftSIMD processor datapath. As the VWR cannot currently be modelled in the Target environment, instead a very simple to model but energy-inefficient register file has been used, with 6 ports (5R + 1W), 48 bit and 16 words. Every access to such a large multi-port RF consumes a very large energy [Ragh09]. That explains why it consumes 8% of the total softSIMD power [Dak11]. However, with the use of the VWR the energy can be reduced by at least a factor 10 [Ragh07]. That will remove the vector VRF contribution from the current power pie results for the SoftSIMD processor.

The register file (RF) for this version of the data-path is a normal register file, since no special grouping of data is needed. The scalar data-path, needs to provide only a very limited performance since the instructions are not executed on multiple data and the frequency of activation is very low. So also the energy contribution is not critical at all. In particular, even with such an unoptimized design the scalar RF uses only 2% of the total power [Dak11]. With some optimization effort on the mapping and scheduling part of the architecture that can easily be reduced further. It currently uses a 3-port register-file with 16 words of 16 bit which is also an overkill in size but which allows an easy scheduling in

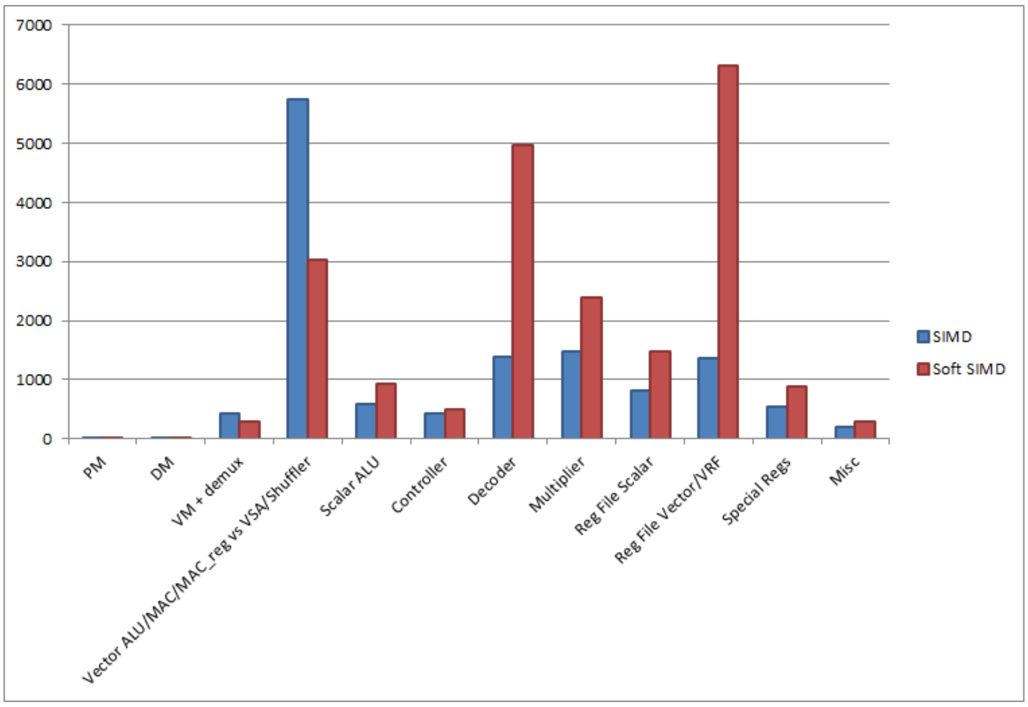


Figure 4.10: Hard-SIMD and Soft-SIMD area comparison per component

the Target environment.

To sum up there are several possible reasons that can explain the results of the comparison, apart from the apparent differences in the architecture:

- The Hard-SIMD handles 64-bit in a 4-way SIMD where the Soft-SIMD 48-bit data in a seemingly 3-way SIMD, but due to the nature of Soft-SIMD it is more flexible and can be also used in different ways.
- Slightly different tool versions used for synthesis
- Soft-SIMD needs Loopbuffers to match SIMD
- Simulated at 100MHz, where the Soft-SIMD could take advantage of a better suited frequency.
- No set limit containing the total critical path in the Soft-SIMD.

The purpose of these comparisons is not to prove one method better than the other, but to observe the impact of the different implementations in energy and power consumption.

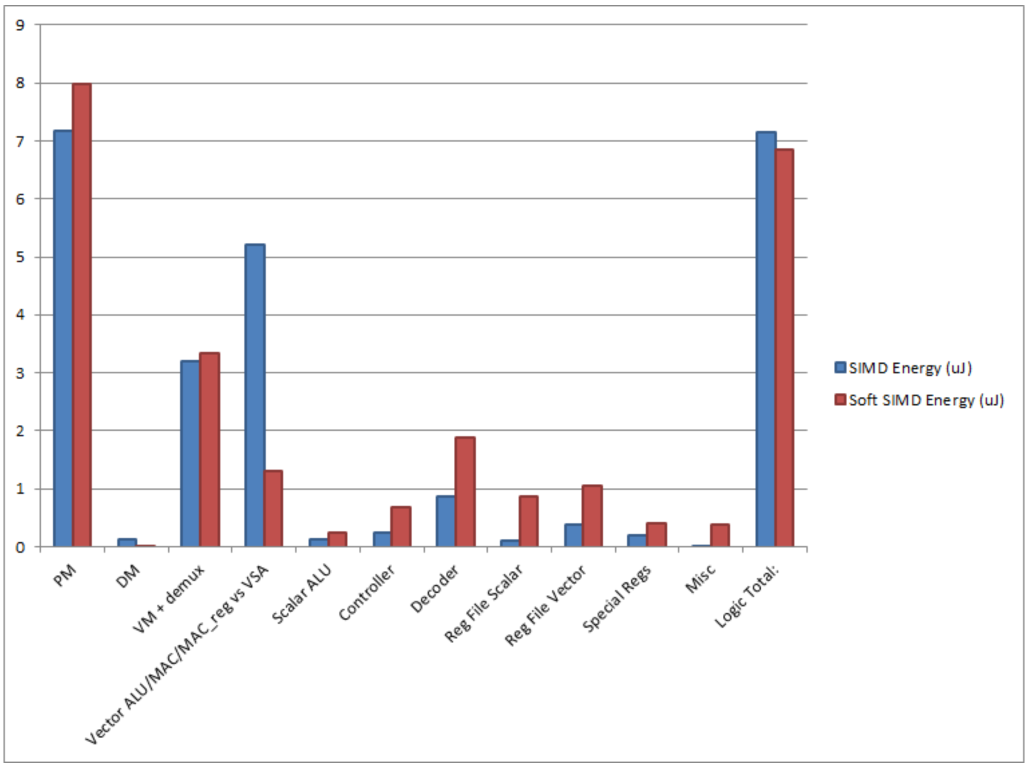


Figure 4.11: Hard-SIMD and Soft-SIMD energy comparison per component

Chapter 5

Modifying the Instruction Set for Energy-Efficiency

A big part of the main focus also includes a series of experimental simulations on the SIMD implementation presented and analysed in the previous chapter (chap. 4). This methodology enables the designer to fully exploit some of greatest advantages of ASIP design, retargetability and easy architectural exploration through iterative simulations. The designer can use the profiling data along with the simulation and power analysis results to come up with several possible solutions and through the automation of ASIPs he also has the capability to try all of them and choose the best suited one.

For the following experimentation, one of the basic *architectural design constraints* is that there should be no change in the basic functionality of the instructions or their total number. Even though techniques that change, merge or remove instructions could prove to the benefit of the design, it is a parameter that would rapidly increase the complexity of the experimentations, and for the purposes of this thesis is deemed unnecessary.

5.1 Analysing the generated control signals for full orthogonality

Before making any changes to the nML code, it is first essential to have an understanding of how the nML code that describes the instruction set of a processor is translated as the decoder into HDL code. One of the initial ideas for potential approaches was to create a very large instruction word divided in different parts for the different functional units (or instruction "families"). This would allow the designer to keep a main control over that family and keep it separate from the rest of the instructions. The extremely compact format exists already, so it is worth going to the other extreme and then trying from scratch to encode and compress the instruction set with various different methods. This data can be used as a basis for a fully orthogonal instruction set, where the designer is able to have full and direct access to the SIMD datapath (see figure 5.1) but as it turns out, that may be promising in theory but impossible to achieve with the current capabilities of the ASIP tools.

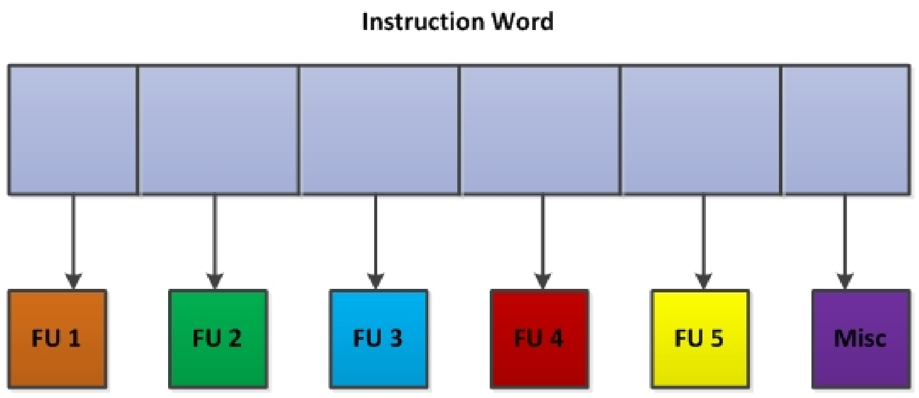


Figure 5.1: A very wide instruction word divided in different parts, each one containing the control bits for a functional unit

Having an instruction word comprised of control bits would virtually remove the need for decoding or at least a great part of it, as the control bits would simply have to be forwarded to the right signals, thus transferring the complexity of the decoder to the compiler that produces the instruction words for the specific processor. Shutting down a functional unit could be as easy as sending an all zeros (“0"s) signal or in some other way that would make use of the don’t care conditions to shut down the functional unit temporarily and additionally reduce the toggling caused by that part of the instruction word.

Analysing the instruction set of the newly developed SIMD processor shows that in the decoder the 16-bit instruction uses 137 control bits to control the memories and the datapath. Naturally, most of the control bits are controlling the execution E1 stage of the pipeline, and around 28 of them the decode stage. Very few bits are used for the E2 stage of the pipeline, since that is only used in the MAC unit and does not require a lot of control bits in the instruction word.

The control bits are analysed and grouped depending on the functional unit or the purpose they served. Controlling their exact encoding in a very wide instruction could prove greatly beneficial for a truly energy efficient instruction set and the experimentation with different encoding.

The instruction control bits are grouped depending on the instruction family that they serve. For example, for an add instruction it is obvious which signals the decoder is driving and to exactly what bit sequence they are matches, since they are generated as a result of an opn rule. The designer can also change that bit sequence in a way that reduces the toggling between subsequent instructions or group them for avoiding having to pay multiple times the same area for similar decoders.

However, as it turns out, at the current version the Target tools may support orthogonality on a subword level but not full orthogonality, in a way that the designer can have complete and direct control over the generated control signals through the instruction bits. The Target tools do not support that degree of freedom in the definition of an instruction set. In case of an orthogonal instruction format the tools allow the construction of different orthogonal sub-classes for the instruction word.

The root cause of the problem lies in the inability of the nML to describe the actual control signals. The tools still consider the instruction as encoded and treat different control signal areas as a single and so they produce the same control signals all over again. They cannot detect that only certain signals are meant to drive certain functional units and ignore the rest. There are transitories that define the inputs and outputs of the functional units in the opn definitions of the nML. Their use is not compulsory, but defining and using them typically results to a more robust design and less errors in compiling the code describing the processor. The whole point of ASIP design is to assist the designer in such matters, and therefore the designer cannot have full control over the generated design, but can always intervene manually on the HDL code generated.

Even though this part of the experiments cannot prove as fruitful as initially expected, the analysis of the control signals assists in getting a better understanding of how the ASIP tools create the decoder and the control signals for each processor design, and how the definition of each instruction encoding can play a small but important part to that. It remains to be seen if such a scheme could actually be energy-effective, since full orthogonality, and subsequently a vast number of control signals can either prove favourable or potentially interfere in a bad way with different components on power-gated or power-downed regions. It might add implementation and verification challenges and overcomplicate the design on later stages of the production.

5.2 Creating a wider instruction

Considering that full orthogonality is not within the potential of the tools, the best approach is to take it step by step in an attempt to widen the instruction width as possible and see the impact of that on energy consumption.

As a first step towards an orthogonal instruction set, the opcode, the operands and the immediate value are rearranged and grouped so that each one has its own specific bit area in the instruction word. That should allow to work more easily and focus on the opcode part. Additionally, it should drastically decrease the complexity of the addressing modes in the decoder, since the operand address or the data has a specified spot in the instruction set.

The resulting instruction word of this idea is shown in figure 5.2 and has a total size of 48 bits:

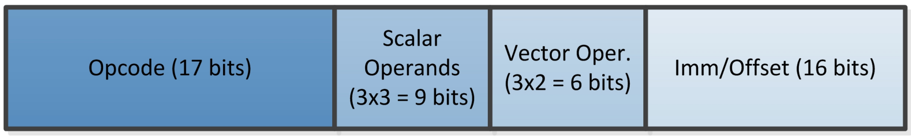


Figure 5.2: The 48-bit orthogonal instruction set

- 17 bits for all of the the opcodes.
- 3x3=9 bits for the three scalar operands.
- 3x2=6 bits for the three vector operands.
- 16 bits for the immediate value.

The new design might be bigger but provides more freedom for various favourable changes and optimizations.

To make sure that the generated decoder would be as simple as possible the unused parts are filled with don't care 'X's to ensure the decoder would not take them into account. Additionally, the opcode part that is common in several instructions is rearranged in order to occupy the same part of the instruction word. That is especially important for the instructions controlling the MAC unit, as it is those instructions that are repeated the most.

5.2.1 Power Results

The results of this implementation are surprisingly good. Even though size of the instruction word used is now three times bigger, the total power consumption is almost 10% reduced compared to the original implementation. As illustrated in figure 5.3, there is significant drop both in internal and switching power.

The total number of toggles throughout the simulation for the same application has a great reduction of 30%. In the individual components the greatest drop comes from the MAC unit, which features a drop of around 30% and the MAC unit is now consuming 32% less power for the same task. Even though the decoder now has a substantially lower toggling activity, it still has roughly the same power consumption.

5.2.2 Area Results

Looking into the area there is an increase of 20% in the area the decoder takes up, which is a very good result considering that the instruction word is now 48

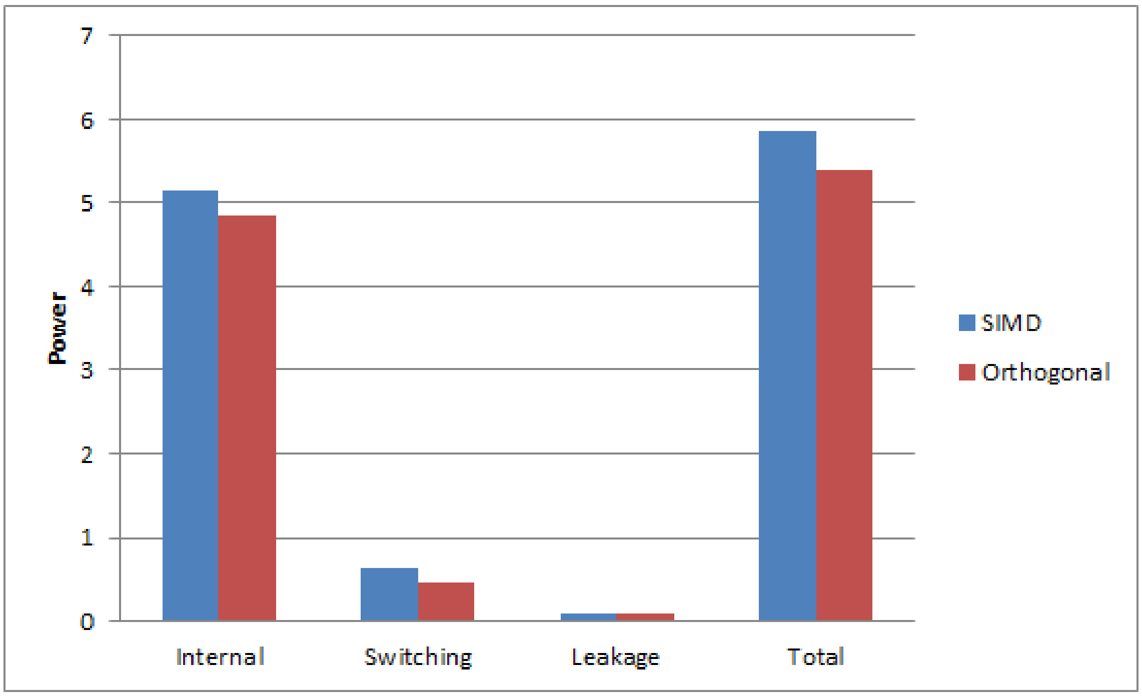


Figure 5.3: Comparison of the average power consumption in uW of the original SIMD (blue) and the orthogonal version (red).

bits long instead of 16 bits, but there is a decrease in other parts of the design, mainly the MAC unit that features a drop of 38%. This leads to an overall area consumption virtually the same between the two implementations. It should be noted that these results are the cell area of the rtl compiler, meaning that they are only an estimation and by no means the final results of the area that the design would require.

5.3 Optimizing the encoding of the instruction set

The 17-bit opcode of the new orthogonal instruction set in many of the instructions includes bits that are unused and can be used for the reducing the decoding of the decoder.

5.3.1 Reducing decoder complexity

The decoder of a processor is defined by the encoding of the instruction set, so the way each of the instructions are defined, grouped and assigned to bit sequences has a great influence on the area and power of the decoder and through that on the whole design. The designer needs to take into account that a sophisticated, complicated instruction set with many instructions compressed into it can be very potent in the tasks it can perform and the available instructions it provides, however a simpler instruction set reduces the complexity of the decoder leading to a smaller hardware footprint that consumes less energy. A simpler instruction set also helps the designer and the retargetability purposes of ASIPs because it allows for fast compiling and test and simulation and additionally makes it easier to update the instruction set to meet the needs of a new task.

Simplifying the decoder

To achieve the goal of a simple effective instruction set there are several ways but not all of them can be combined and they don't always lead to good results. The method followed should be one that matches the target application domain of the processor.

There are various ways to reduce decoder complexity[Target nML]. But to reduce it one first needs to comprehend what it actually is that builds up the complexity of an instruction set. To follow a common example, assume there are three operations on an ALU with the following patterns:

add	$\rightarrow$	000xxxxx00xx
sub	$\rightarrow$	000xxxxx01xx
or	$\rightarrow$	000xxxxx10xx

All the units that work in parallel with the ALU (like a multiplexer connecting the output of the ALU to a bus or a memory) are only enabled for the instructions that obey the following pattern:

000xxxxx0xxx
000xxxxxx0xx

The same procedure needs to be followed to create patterns for all the potential parallel functions that would need to make use of the ALU and the same method has to be applied for all the rest of the modules in the design. So if another ALU operation would require the two rightmost bits of the instruction to be enabled then the new enabling conditions would be:

000xxxxx0x00
000xxxxxx000

The problem arises when the compiling tools for the decoder are trying to match the enabling conditions for a lot of instructions that would require enabling control to that functional unit. With each new instruction being introduced, the complexity of the decoder rises at an exponential rate until it virtually explodes, resulting in a big or even unmanageable hardware footprint. Therefore the designer needs to apply optimal encoding in the bit sequences of the instruction set encoded to keep the complexity at affordable levels.

nML Complete Image Option

Target also provides the *complete_image* option. This helps to indicate that an nML AND rule specifies a complete encoding, and that its fields are orthogonal. For example:

```
opn my_rule (a : A, b : B) complete_image
{
  action { a; b;}
  image : "000"::a::b;
}
```

This option helps to point out to the nML front-end tools that there is orthogonality between the instruction parts that specify complete register transfers (from either the register file or the memories to again either the register file or the memories as explained in section 1.1.2).

When this rule is active, only the first part of the opcode "000" is checked, even if A and B do not have complete encodings. Therefore all the bits that are not opcode bits are treated as don't care for rules higher in the nML hierarchy. ..which unfortunately did not seem to make any difference at all in the design, and during the decoding of A the nML tool will consider B to have a complete encoding and vice versa. This can assist the developer reduce the decoder complexity when there are incomplete definitions, especially when designing a VLIW processor.

Nonetheless, despite the various attempts to employ this option in our experiments with the processor, the generated rtl code is the same and the tools are still considering the instruction set to be a non-orthogonal one.

nML code quality

The processor designer should be fully aware of the impact that the way in which he chooses to describe the instruction encoding has in the transition from nML grammar to the generated HDL code. Even though the nML grammar provides a plenty of options that are syntactically correct, the repercussions of a a badly structured instruction encoding can be severe on the area and power budgets of the decoder and subsequently to the whole design.

The nML code quality has to be optimal to the match the targeted processor specifications. Any arbitrary additions to the nML code should be strongly avoided and the designer by following the methodology previously explained can quickly develop the required skills through trial and error to avoid the common pitfalls.

5.3.2 Exploiting methods for energy efficient architectural design with ASIP tools

When introducing ASIPs, it was mentioned that one of their key features is the capability to make quick adjustments to the design and that in each of the stages of the ASIP design the developer has the choice to generate debugging information. Using these data can prove vital for the full utilization of the design for a specific application or application domain.

As analysed on an earlier chapter, Chess is a retargetable C compiler that translates C source code into machine code for the target processor. The Target tool-suite also includes Checkers, a retargetable instruction-set simulator (ISS) that produces a cycle and bit accurate simulator for the target processor.

By combining the Chess tool to produce the machine code of the targeted application and Checkers tool for simulating and debugging the execution, the designer is able is to extract various useful data about the statistical usage of the instructions and the critical functions that will need to run on the processor, and therefore can focus and experiment with them. It should be noted that for typical DSP applications, the code is usually characterized with a 20/80 rule, meaning that 80% of the processor clock cycles are spent on a specific 20% part of the code, consisting of DSP kernels [Goos04].

Profiling and execution tracing instructions and data accesses

Using the Checkers ISS tool, profiling and execution tracing data can be extracted for a particular application. These profile data contain information about which are the parts of the program where most of the cycles are spent. There are additional profiling options available for instruction classes, primitive operations, functional units and storage accesses that can be very helpful in focusing and pointing out the bottleneck of the design.

The execution trace shows the call and return history during the simulation. That can be really useful in investigating the overall execution of the program, along with which functions are used in an application and how many times each function is called. The storage profiling provides information about the access history of the memories and the register file.

Spatial and temporal locality exploitation

The nML language combined with the Chess and Checkers tools are ideal for power-conscious architectural design, as they offer an architectural scope wide enough to allow for experimentation with many different techniques for low power consumption. A simple and common way to reduce the total energy consumed for a task would be to reduce the total number of processor cycles required for the execution. That can be managed by exploiting instruction-level parallelism, bundling several instructions into a single one, employing special purpose registers or highly encoded instruction sets.

By taking advantage of the ASIP tools, the designer can make changes in the nML code and then very soon have the new HDL code and the profiling results of simulating the new design. So by using the available profiling and execution trace data the designer can experiment in many different directions and find the changes that can lead to an optimal instruction encoding and an overall energy efficient design.

As explained on chapter 2.4, *Huffman encoding* can be used to take advantage of the *temporal locality* of the instructions in order to minimize the cost of subsequent instructions used in the most common loops. In a loop that is executed repeatedly hundreds or even million of times in a single application reducing the toggle of subsequent instructions can prove greatly beneficial.

In a similar way the *spatial locality* of the memories can be investigated, and the nML code can be build to suit the needs of the targeted application in a way that the memory hierarchy can be used to its fullest especially in the most common loops without repeating the same memory transfers for different iterations over and over again. Apart from changing the instruction mechanics, the instruction and data memories along with the register file can also be changed to match the application.

5.4 Final implementation

The aforementioned methods are employed to the orthogonal processor that was developed. According to the profiling results, from the 47 instructions that the filter program was compiled into, 9 of them are used 96,5% of the time. These instructions are mostly comprised of vector load/store instructions and vector MAC instructions. In more detail there are 3 vector loads, 1 vector store, 3 multiply-accumulate instructions, 1 accumulator initializing instruction and 1 accumulator shift instruction for storing the right part of the final result.

By making full use of the available ASIP tools, there are many different experimental changes that can be conducted. There are many attempts to simplify the decoding procedure of the instructions in an effort to use the spare bits of the instruction word in a way that would help decrease the decoder component cost, but that proves only slightly favourable for the whole design at best. It should not be forgotten that improving the decoder can sometimes lead to an overhead for other components thus leading to an overall greater energy cost for the whole design.

On the other hand, what proves evidently fruitful is the attempts to apply different encodings to the instruction set, to reconstruct it in a way that reduces both the power consumption of the most commonly used instructions and the power consumed due to the toggling between the subsequent instructions of the main loop.

In order to have a metric on the toggling from one instruction to another, *Hamming distance* is introduced. The Hamming distance between two bit vectors is simply the number of bits that are different between the two bit vectors. So the aim of the encoding to reduce the toggling would be to reduce the Hamming distance of each instruction as much as possible with the one that follows it, or do the same procedure for all the most commonly used instructions to any of the others.

A simple encoding following the principles of Huffman encoding is employed to the instruction set with a focus on the instructions that make up for the main loop of instructions, but always avoiding any encoding that might lead to problematic HDL. The ASIP tools are very strict in this part, and do not allow the designer to the make any mistakes or clerical errors in the encoding.

5.4.1 Final results

The results from this final implementation as shown on the following figures, are very good. Without any major changes to the instructions themselves, only the bit encoding of the instruction grouping hierarchy and the instructions themselves, the power consumption of the decoder features a drop of 12% as compared to the previous orthogonal design and the whole design has a 8% drop.

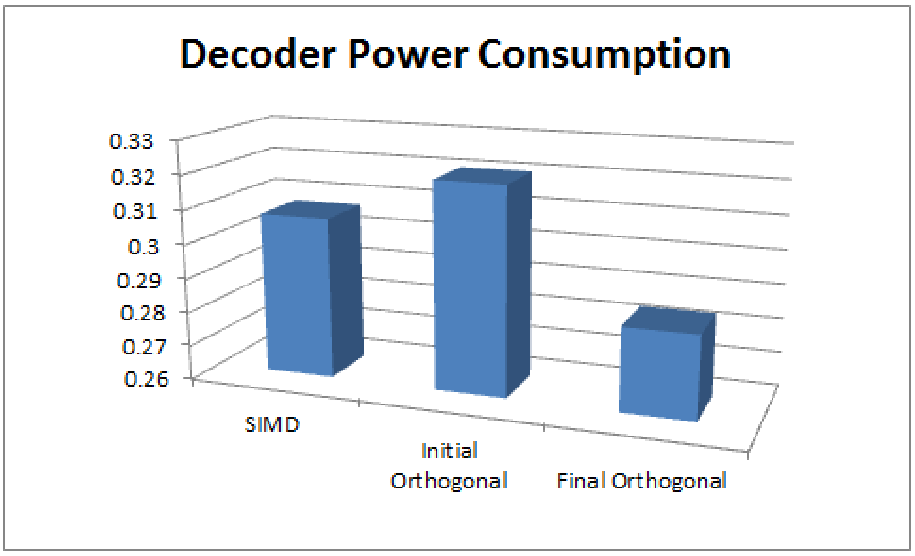


Figure 5.4: Decoder component power consumption of the SIMD, first and final orthogonal implementations

Compared to the first SIMD design, as shown in figures 5.4 and 5.5 there is now an 8% drop in the power consumption of the decoder and 15% drop in

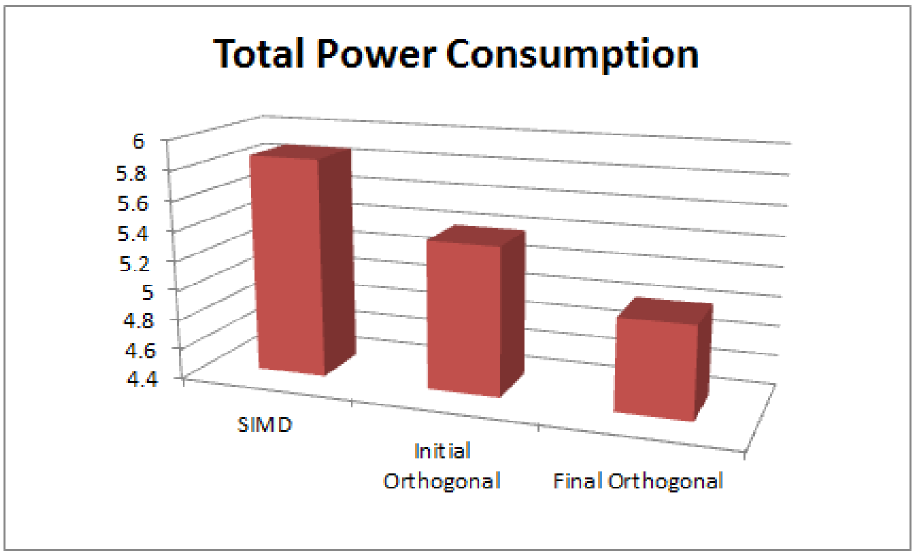


Figure 5.5: Total power consumption of the SIMD, first and final orthogonal implementations

the power consumption of the total design. Which is more than was expected, especially considering that the changes made are exclusively on the encoding of the instruction set.

Looking into the drop in the toggling activity, as a means to reduce the energy consumption there is a significant 18% drop in the decoder as compared the first orthogonal implementation and a 65% drop as compared to the original SIMD implementation, even though in the SIMD the instruction word is only 16 bits long.

The component comparison chart (figure 5.6) shows the statistics in the first and the final orthogonal implementation are quite different from the ones in the CoolBio or the SIMD implementation. Once again the power consumption of the memories -that would be dominant- is left out, in order to focus on the components of the core. Despite the aforementioned drop in the power consumed by the decoder and the whole design, there is only little change in the decoder as compared to the other components, with the vector MAC unit taking up the largest part of the power consumption of the core.

In terms of area, as illustrated in figure 5.7, there is little difference between the two orthogonal versions, however there is a small drop in the cell area required for the final orthogonal design and there is also a similar drop in the net area.

There are also other experiments conducted with different instruction word sizes or more radical changes. But the code generated cannot always be properly checked and synthesized by the HDL compiler tools due to timing violations or compiling errors. Especially when using arbitrary bit lengths for the instruction word other than the usual ones in powers of two.

5.4.2 Conclusions

To sum up, as analysed on this chapter there is a lot to gain in the research that of how the length of the instruction word can be changed to suit the application domain. The ASIP tools offer retargetability and easy architectural exploration that be used for iterative modifications and simulations on a design, in an attempt to find the best solution for the targeted application domain.

According to the initial assumption, an orthogonal instruction word with a width of 48 bits instead of the original 16 bits is build and experimented on to find the capabilities it can offer. By taking into account all of the parameters (area, performance, energy consumption, available hardware), the design can be changed according to the respective needs of the time with great gain and little effort, as long as certain guidelines are followed.

The results, as shown in the previous figures, reveal a substantial gain in energy consumption, thus proving that the original idea was a success. Having a larger instruction word provides the essential space for optimizations in the instruction encoding that can greatly reduce the energy consumption of the decoder and the whole design.

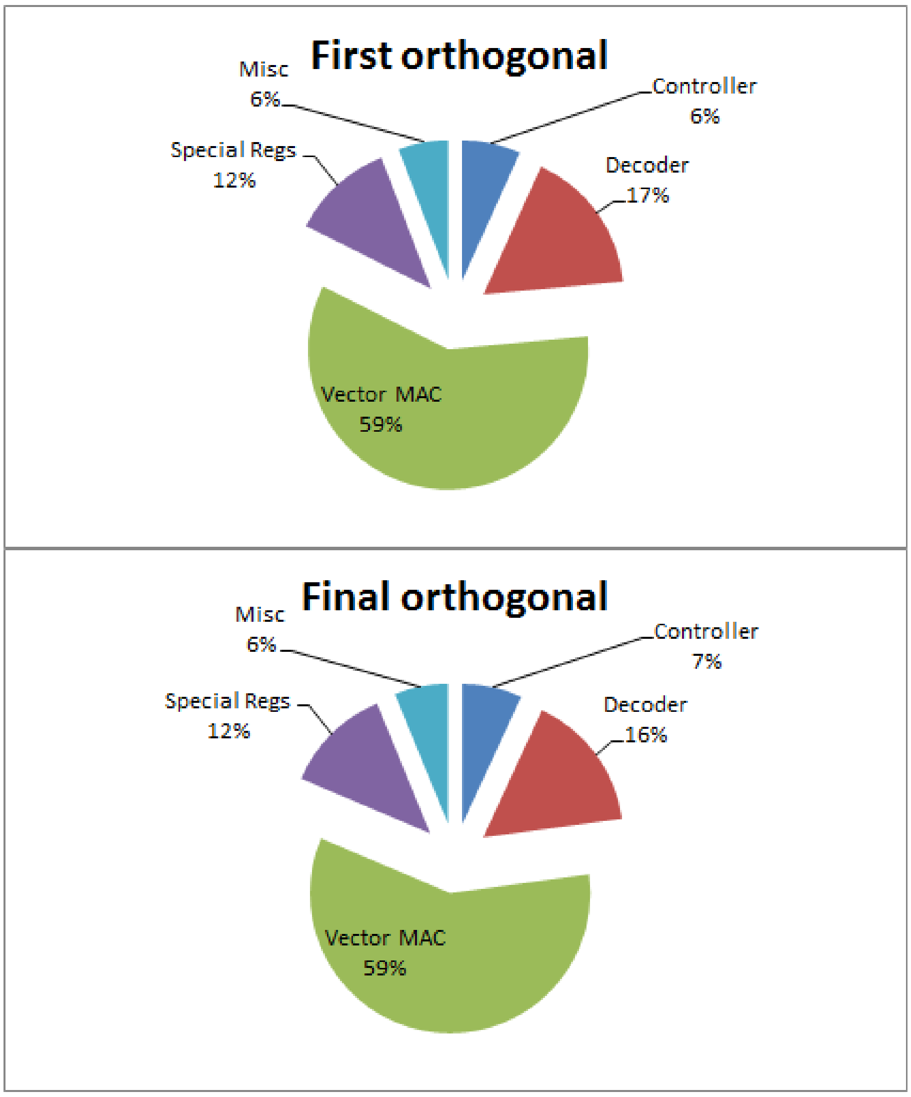


Figure 5.6: Power consumption for every component in the two orthogonal implementations

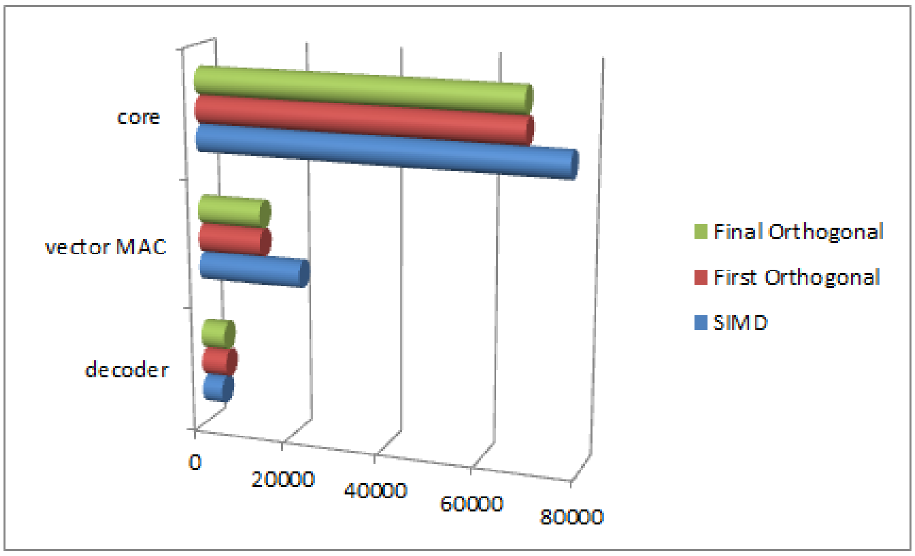


Figure 5.7: Cell area required for the SIMD and the two orthogonal implementations

Chapter 6

Conclusions

The purpose of this study is to find a method that can utilize the available advanced ASIP tools -like Target's tool flow- to produce a low-energy processor for a specific application domain. For that purpose, ASIPs are developed and used for their distinctive ability to combine energy efficiency, flexibility and performance. It should not be forgotten that in today's embedded systems energy consumption is one of the greatest problems. However, since technology advances rapidly, some of the solutions to problems that designers used to face have to be revisited to assess if they can be solved in a move effective way.

The available ASIP tools offer sophisticated tools that enable the development of an ASIP processor in only a fragment of the time that would be required for the design of a normal processor. They offer a full suite from processor architecture design, instruction set description and operation definitions, to application simulation and HDL generation. The available simulation capabilities can be combined with the profiling to investigate different implementations through iterative changes to the original code with the aim to achieve the optimal for the targeted application domain.

6.1 The quest for the golden ratio

As mentioned before, *processor design is a game of many trade-offs*. So in order to achieve the so-called golden ratio, the ideal balance for a processor architecture, a designer would need to figure out the influence of the different parameters (i.e. energy, power, performance, flexibility) on each of the components and the interconnections of a processor and choose the trade-offs that make it as efficient as possible within the technical characteristics defined.

It is a usual mistake to think that power and energy consumption are one and the same. Power efficiency does not necessarily guarantee energy efficiency. In embedded systems nowadays, energy is the decisive factor that can actually make a processor successful. Low power consumption or high performance alone are not enough, since the two seem to be in contrast to the other. A processor with very low power consumption is commonly the goal of the design, however this usually leads to very low performance (or vice versa), so it takes up more energy for the processor to perform a task. Therefore, the designer of a processor has to make a compromise and keep the balance between power and performance, depending on the application at hand. Thankfully, applying modifications on ASIPs is much easier and faster compared to normal processor design.

There are several techniques for ultra low-power processor design that be effectively applied on different levels (compiler, architectural or circuit), like dynamic voltage scaling, clock gating or encoding. Each one has its own advantages and disadvantages and does not always lead to energy efficient design.

6.2 Memory efficiency

It is usually taken for granted that the smaller the memory in a processor, the better. But what about the overhead that has on the rest of the components in a design. It is exactly for this reason, that it is investigated whether a larger instruction word can actually be better for the design, allowing the designer to use the extra bits of the instruction word to tailor the instruction set to the application domain and then encode it once again in a more effective way, despite the extra overhead that the program memory would need.

Studies on the different commercial SRAM memories that are available show that while bigger memory arrays (comprised of smaller subarrays) tend to offer more storage for the area that they require, their power consumption (relative to performance) tends to increase at a faster rate as the word size and the block capacity increases. That means that while smaller memories may be best in power efficiency, larger memories can offer multiple times more memory storage for relatively smaller area, which in turn has a direct impact on power. So the answer to the energy efficiency problem lies somewhere in between.

A designer can take advantage of these findings, and use relatively larger (but not too large) memories with word sizes of up to 64-bits that can offer more storage with relatively smaller power cost.

6.3 ASIP instruction-set architecture

The instruction-set architecture is the most important characteristic and what defines a processor. That is even more the case in ASIPs where the software part of the hardware/software co-design tools have to functionally adapt to this instruction-set and support it.

6.3.1 Hard-SIMD and Soft-SIMD

One of the most important processor architectures of last decades is SIMD. It allows the exploitation of instruction-level parallelism. There are several different approaches to implementing an SIMD, mainly in hardware or software. Hardware SIMD relies on multiple functional units in hardware while software SIMD transfers that complexity to the compiler software and the decoder.

A hardware SIMD developed for this thesis is compared to a simple scalar processor and a software SIMD implementation with the aim to investigate their energy and area efficiency. A bioimaging application based on a Gauss filter is used as the benchmark. The results show that both SIMD implementations fare much better than the scalar processor, mainly in terms of performance. Getting the job done faster with low power to performance ratio, results to smaller overall energy consumption.

Comparing the hard and soft-SIMD proves more tricky, as they are totally different implementations, but what is the main focus here is energy consumption. The hard-SIMD requires slightly less energy for the biotechnology benchmark, with a lower cell count as well. The hard-SIMD features four power and area costly multipliers to perform MAC operations with a 16-bit instruction-set and an overall simple design, while the soft-SIMD is heavily optimized with a sophisticated vector register file, a vector shift-add unit (substituting a MAC with less energy) and an 80-bit instruction word that can adapt to different subword sizes. Looking into the energy component breakdown (figure 4.11), it is obvious that the complexity derived from the additions in the soft-SIMD is also transferred to the rest of the components of the processor, especially the decoder which is overburdened with the task of decoding an 80-bit word. So from the energy efficiency perspective, the hard-SIMD with its simpler and cleaner design, seems to be preferable as it is much faster to develop and debug, and also provides space for taking advantage of the ASIP tools through experimental optimization.

6.3.2 Orthogonality and optimal word size

The hard-SIMD implementation had its instruction-set modified from 16-bits to 48-bits with an orthogonal approach in mind. The initial idea was for the instruction-set to be much wider and fully orthogonal, being able to fully control the generated signals and have an instruction-set that provides all operations to addressing mode combinations, but that is rendered impossible by the ASIP tools available. Instead a 48-bit orthogonal instruction-set is developed. This includes a 17-bit opcode, 9 bits for the three scalar operands, 6 bits for the three vector operands and 16 bits for offsets and immediate values.

This design, even though it requires a four times bigger program memory, proves to be 10% more energy efficient than the original hard-SIMD design, due to reducing the power consumption of the decoder, the MAC unit and the rest of the components.

There are also several experiments conducted on how to reduce the complexity of the decoder through the instruction-set encoding. That is achieved by using unused bits in the 17-bit opcode, fine tuning the nML to improve the quality of the HDL generated and experimenting with different encoding schemes or instruction word sizes. The ASIP tools provide various instruction trace and profiling tools that can prove very helpful in combination with the relatively small time required, in order to exploit the different options in spatial or temporal and find the best suited for the design. The only problem here is that there needs to be an automated way to do this and verify the design on each of the stages from processor nML design to synthesis and power benchmarks, as well as extracting the results in a user friendly way. This is solved with the use of several scripts in various programming or scripting languages.

The final implementation is a result of a combination of these methods, keeping the main constraint of maintaining the instruction functionality and number the same. It shows that mostly by using a different encoding for the instructions in the loop kernels which are the ones that are executed the majority of the time, there can be a major drop in the toggling activity of the decoder and the memories, which in turn contributes to a drop of 8% as compared to the initial orthogonal design and 15% compared to the SIMD design.

It is obvious that the instruction-set encoding and its width can have a major role on the energy and performance of a processor. By using the ASIP tools and their retargetability, the design can be tailored to perform effectively in an application domain. The size of the instruction word defines the relation of the complexity between the memories and the design. A smaller instruction word requires a small memory and transfers the complexity to the decoder, while a larger word has a simpler decoder but requires a larger memory. However, there are times when reducing the toggling activity of the instructions can be more important than the size of the instruction word.

Combining the results from this experiments on the ASIP implementations along with the data from memory efficiency, it is suggested that since energy is now one of the most defining features of a design, a processor and especially an ASIP can profit from having a larger instruction word size (that is not highly compressed), which reduces the complexity (and thus the area and power required) by the decoder and provides the option of many possible optimizations without incurring a big overhead on the memories and the processor components. Nevertheless, the increase in the size of the instruction word should be kept to manageable levels because of the power required by memories above 64 bits. It is up to the designer to choose the instruction word size that is the ideal for every situation.

6.4 Future work

The ASIP tools are ideal for the design of DSP co-processors for Multiprocessor System-on-Chip (MPSoC). Application domains including audio/video processing, biotechnology, encryption or baseband systems for next-generation wireless modems are good candidates for an ASIP implementation, where having a dedicated co-processor with great performance and low energy cost per task is crucial for the success of the system. It is in these cases that reconfigurable architectures like ASIPs are most effective, because of their ability to adapt with new instructions to new demands.

There should also be an automated procedure that can help use the potential of iterative experiments for the investigation of ASIP design, in order to find the optimal conditions (i.e. instruction-set encoding).

Even though the Target tool flow is still developing at a fast rate, Target should try to emphasize some features which are essential for designers to have more control over their design as their processor description goes through the various stages of ASIP flow and is simulated, until it is finally translated into hardware.

Bibliography

[ARM] ARM Holdings plc, *November 2011,*
`www.arm.com`

[Artes10] A. Artés García, "Energy Impact of Loop Buffer Schemes for Embedded Systems", *Master Thesis, Universidad Complutense de Madrid,* 2010.

[Barat03] F. Barat, M. Jayapala, T. Aa, R. Lauwereins, G. Deconinck, and H. Corporaal, "Low power coarse-grained reconfigurable instruction set processor", *Field-programmable logic and applications,* 2003.

[Beni02] L. Benini, D. Bruni, a Macii, and E. Macii, "Hardware-assisted data compression for energy minimization in systems with embedded processors", *Proceedings 2002 Design, Automation and Test in Europe Conference and Exhibition,* 2002.

[Bonn08] T. Bonny, J. Henkel, "Efficient code compression for embedded processors", *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, vol. 16,* 2008.

[Braun04] G. Braun, A. Nohl, W. Sheng, J. Ceng, M. Hohenauer, H. Scharwächter, R. Leupers, and H. Meyr, "A novel approach for flexible and consistent ADL-driven ASIP design", *Proceedings of the 41st annual conference on Design automation - DAC,* 2004.

[Burd00] T. Burd, T. Pering, and A. Stratakos, "A dynamic voltage scaled microprocessor system", *Solid-State Circuits,, vol. 35,* 2000.

[Catth88] F. Catthoor, J. Rabaey, G. Goossens et al, "Architectural strategies for an application-specific synchronous multiprocessor environment", *Acoustics, Speech and Signal Processing, IEEE Transactions on, vol. 36,* 1988

[Catth10] F. Catthoor, P. Raghavan, A. Lambrechts, M. Jayapala, A. Kritikakou, and J. Absar, "Ultra-Low Energy Domain-Specific Instruction-Set Processors", *Springer,* 2010.

[Chat07] A. Chattopadhyay, D. Zhang, D. Kammler, and E. Witte, “Power-efficient Instruction Encoding Optimization for Embedded Processors", *20th International Conference on VLSI Design held jointly with 6th International Conference on Embedded Systems (VLSID '07),* Jan. 2007.

[Corma87] G. V. Cormack and R. Horspool, “Data compression using dynamic Markov modelling”, *The Computer Journal, vol. 30*, 1987.

[Corme01] Cormen, T. H., Leiserson, C. E., Rivest, R. L., Stein, C. “Introduction to algorithms”. *The MIT press*. 2001.

[Dak11] S. Dakourou, “Optimized SIMD architecture exploration and implementation for ultra-low energy processor architectures”, *Master Thesis, Dept. of Computer Eng. and Informatics, Univ. of Patras and IMEC/Holst Centre*, 2011

[DeMan86] H. De Man, J. Rabaey, P. Six, and L. Claesen, “Cathedral-II: A Silicon Compiler for Digital Signal Processing”, *Design Test of Computers IEEE*, 1986.

[DeMan88] H. De Man, J. Rabaey, J. Vanhoof, G. Goossens, P. Six, and L. Claesen, “CATHEDRAL-II-a computer-aided synthesis system for digital signal processing VLSI systems”, *Computer-Aided Engineering Journal, vol. 5* , 1988.

[Emmet00] F. Emnett, "Power reduction through RTL clock gating ", *Synopsis User Group (SNUG) Conference*, 2000.

[Geur05] W. Geurts, G. Goossens, D. Lanneer, and J. Van Praet, “Design of application-specific instruction-set processors for multi-media, using a retargetable compilation flow”, *Proceedings of Global Signal Processing (GSPx) Conference, Target Compiler Technologies, Citeseer*, 2005.

[Glok04] T. Glökler and H. Meyr, “Design of energy-efficient application-specific instruction set processors”, *Springer Netherlands*, 2004.

[Goos87] G. Goossens, J. Rabaey, J. Vanderwalle, and H. De Man, “An efficient microcode-compiler for custom DSP-processors”, *IMEC Laboratory, B-3030 Leuven, Belgium*, 1987.

[Goos04] G. Goossens, D. Lanneer, and P. Dyrtrych, “Design of Low Power Processor Cores using a Retargetable Tool Flow”, *retarget.com*, 2004.

[Goud99] L. Goudge and S. Segars, “Thumb: reducing the cost of 32-bit RISC performance in portable and consumer applications”, *COMPCON 1996. Technologies for the Information Superhighway Digest of Papers. IEEE Comput. Soc. Press*, 1999.

[Hatt95] E. Hatton, "SAMC-efficient semi-adaptive data compression", *Proceedings of the 1995 conference of the Centre for Advanced Studies on Collaborative research, p. 29*, 1995.

[Henn06] J. L. Hennessy and D. A. Patterson, "Computer Architecture: A Quantitative Approarch", 4th Edition, *Morgan Kaufmann*, 2006.

[Hoel72] P. G. Hoel, S. C. Port and C. J. Stone, "Introduction to stochastic processes", *Houghton Mifflin Company*, 1972.

[Huff52] D. Huffman, "A Method for the Construction of Minimum-Redundancy Codes", *Proceedings of the IRE*, 1952.

[Katev11] M. Katevenis, G. Passas, CS-534 Lecture Slides, *Computer Science Department, University of Crete*

[Kaxir08] S. Kaxiras, M. Martonosi, "Computer Architecture Techniques for Power-Efficiency", *Synthesis Lectures on Computer Architecture, Morgan & Claypool Publishers*, 2008.

[Keat07] M. Keating, D. Flynn, and R. Aitken, "Low power methodology manual: for system-on-chip design", *Springer*, 2007.

[Kiss97] K. D. Kissell, "MIPS16: High-density MIPS for the Embedded Market". *Silicon Graphics MIPS Group*, 1997.

[Krit09] A. Kritikakou, "Low cost low energy embedded processor for online biotechnology monitoring applications", *Master's thesis, Dept. of Computer Eng. and Informatics, Univ. of Patras and IMEC*, 2009.

[Leka98] H. Lekatsas and W. Wolf, "Code compression for embedded systems", *Proceedings of the 35th annual Design*, 1998.

[Leka99] H. Lekatsas and W. Wolf, "SAMC: a code compression algorithm for embedded processors", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, vol. 18*, 1999.

[Leka00] H. Lekatsas and W. Wolf, "Arithmetic coding for low power embedded system design", *DCC*, 2000.

[Leup00] R. Leupers "Code optimization techniques for embedded processors: methods, algorithms, and tools", *Springer*, 2000

[Lin04] C. H. Lin, X. Yuan, and W. Wolf, "LZW-based code compression for VLIW embedded systems", *Proceedings Design, Automation and Test in Europe Conference and Exhibition*, 2004.

[Liu08] D. Liu, "Embedded DSP processor design: application specific instruction set processors", *Morgan Kaufmann*, 2008.

[Maha05] N. Mahapatra, J. Liu, and K. Sundaresan, “A limit study on the potential of compression for improving memory system performance, power consumption, and cost”, *J. Instruction-Level, vol. 7,* 2005.

[Mare06]
H. Maréchal, “ASIP design methodology with Target's Chess/Checkers retargetable tools”, *Proc. Intl. Signal Processing Conference, Santa Clara,* 2006.

[Mish08] P. Mishra and N. Dutt, “Processor description languages: applications and methodologies”, *Morgan Kaufmann,* 2008

[Morg07] P. Morgan, R. Taylor, “ASIP Instruction Encoding for Energy and Area Reduction”, *44th ACM/IEEE Design Automation Conference,* 2007.

[Mutoh95] S. Mutoh, T. Douseki, Y. Matsuya, T. Aoki, S. Shigematsu, and J. Yamada, “1-V power supply high-speed digital circuit technology with multithreshold-voltage CMOS”, *Solid-State Circuits, IEEE Journal of, vol. 30,* 1995.

[Parhi99] K. Parhi, “VLSI digital signal processing systems: design and implementation”. *Wiley-India,* 1999.

[Piguet01] C. Piguet, P. Volet, J. M. Masgonty, F. Rampogna, and P. Marchal, “Code memory compression with online decompression”, *Solid-State Circuits Conference, 2001. ESSCIRC 2001,* 2001.

[Piguet06] C. Piguet, “Ultra-low power processor design”, High-performance energy-efficient microprocessor design, *Springer,* 2006.

[Psy10] G. Psychou, “Optimized SIMD scheduling and architecture implementation for ultra-low energy bioimaging processor”, *Master's thesis, Dept. of Computer Eng. and Informatics, Univ. of Patras and IMEC,* 2010.

[Raba03] J. Rabaey, “Digital Integrated circuits: a design perspective”, *Prentice-Hall,* 2003.

[Ragh07] P.Raghavan, A.Lambrechts, M.Jayapala, F.Catthoor, D.Verkest, H.Corporaal, “Very wide register: an asymmetric register file organisation for low power embedded processors”, *Proc. 10th ACM/IEEE Design and Test in Europe Conf.,* Nice, France, April 2007.

[Ragh09] P.Raghavan, A.Lambrechts, M.Jayapala, F.Catthoor, D.Verkest, “EMPIRE: Empirical Power/Area/Timing Models for Register Files”, *Microprocessors and Microsystems J.* Feb. 2009.

[Roev04] H. Roeven, J. Coninx, and M. Ade, "CoolFlux DSP-The embedded ultra low power C-programmable DSP core", *in Proc. Intl. Signal Proc. Conf.GSPx*, 2004

[Synopsys] Synopsys, *November 2011*
`www.synopsys.com`

[Target] Target Compiler Technologies, *November 2011*
`www.retarget.com`

[Target nML] The nML Processor Description Language 11R1, *Target Compiler Technologies*, March 2011.

[Tensilica] Tensilica, *November 2011*
`www.tensilica.com`

[Weste11] N. Weste and D. Harris, "CMOS VLSI Design: A Circuits and Systems Perspective", *Addison-Wesley*, 2011.

[Wolfe88] A. Wolfe et al., "The white dwarf: a high-performance application-specific processor", *ACM SIGARCH Computer Architecture News, vol. 16*, 1988.

[Wolfe92] A. Wolfe and A. Chanin, "Executing compressed programs on an embedded RISC architecture", *ACM SIGMICRO Newsletter, vol. 23*, 1992.

[Xie07] Y. Xie, W. Wolf, and H. Lekatsas, "Code Decompression Unit Design for VLIW Embedded Processors", *IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 15* 2007.

[Xu04] Xu, X., Clarke, C. T., and Jones, S. R. "High performance code compression architecture for the embedded ARM/THUMB processor". *Proceedings of the 1st Conference on Computing Frontiers*. ACM. 2004.

[Zhang08] D. Zhang, A. Chattopadhyay, D. Kammler et al., "Power-efficient Instruction Encoding Optimization for Various Architecture Classes", *Journal of Computers, vol. 3* 2008.