

## An Integrated System for Rapid Prototyping of High Performance Algorithm Specific Data Paths

D. C. Chen, L. M. Guerra, E. H. Ng, M. Potkonjak<sup>†</sup>,  
D. P. Schultz, and J. M. Rabaey

*Department of Electrical Engineering and Computer Sciences  
University of California, Berkeley, CA 94720*

<sup>†</sup> *C&C Research Labs. NEC USA, Princeton, NJ 08540*

### Abstract

A system has been developed which targets the rapid prototyping of high performance data computation units which are typical to real-time digital signal processing applications. The hardware platform of the system is a family of multiprocessor integrated circuits. The prototype chip of this family contains 8 processors connected via a dynamically controlled crossbar switch. With a maximum clock rate of 25 MHz, it can support a computation rate of 200 MIPS and can sustain a data I/O bandwidth of 400 MByte/sec. An assembler and simulator provide low-level programmability of the hardware. A compiler which takes input described in the high-level data flow language Silage [21], and performs estimation, transformations, partitioning, assignment, and scheduling before generating assembly code, provides an automated software compilation path.

## 1 Introduction

In real-time digital signal processing systems, data often enters or leaves the computation intensive parts at small integer multiples of the clocking interval. Because traditional microprocessor-based architectures are inadequate to meet the high throughput requirements of these systems, clusters of dedicated data paths, hard-wired to closely match the algorithmic data flow, are often used [2]. These architectures typically contain multiple and concurrently operating data path pipelines, and examples can be found in image and video processing [26, 17, 27, 20, 16] and speech recognition systems [15, 17, 14].

It is of great interest to rapidly implement and/or prototype such architectures since this results in both shorter design cycles and fewer and more correctable errors. High-level synthesis for high speed DSP systems is one approach to the problem, e.g. [19, 25, 18, 8]. Other approaches involve multiprocessor configurations based on commercially available DSPs [1], video signal processors or VSPs [31, 28, 12], and FPGAs [9, 11]. Because the aforementioned approaches tend to lack in the areas of performance, flexibility, or interconnect bandwidth, we propose a novel approach using reconfigurable data paths: the initial concept, PADDI, was proposed in [5, 4]; circuit design details of a prototype chip, can be found in [6].

In this paper we will present a complete system for the rapid prototyping of high performance algorithms onto the PADDI architecture. First, the architecture will be presented from the assembly

Datapath	Arithmetic	Operators	Control	Data IO
pipelined, parallel, heterogeneous communication	8-18b unsatd 8-24b 2sc saturation	+ - shift 1/2 .. 1/128 compare min, max acc, mult	local branch global branch	8-112 IP 8-51 OP

Table 1: General Characteristics of Benchmark Set

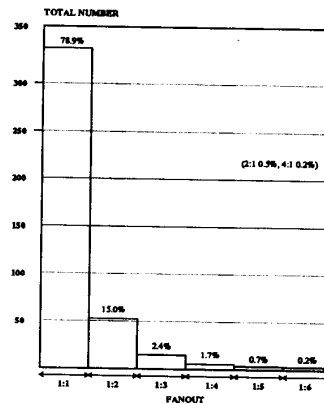


Figure 1: Total Number of Arcs vs. Arc Type

language programmer's perspective. We will also discuss the design methodology which has been employed to obtain high performance. In addition, key architectural features that are exposed to the assembly language programmer and compiler for performance optimization will be discussed. The assembler and simulator which have been developed to enable user assembly language programming will then be covered. Finally, the CAD environment and software tools being developed for automatic compilation from a high-level language [21] will be described.

## 2 Architectural Overview

The PADDI architecture was developed by analyzing a set of real-time digital signal processing benchmarks drawn from image and video processing and speech recognition to determine their computational and I/O requirements. The general characteristics of the benchmark set are presented in Table 1 (for a complete listing of the benchmarks, refer to [3]).

Figure 2 shows the total numbers of each type of operation present in the benchmark set; the operations which are supported in the hardware were chosen based on these results.

Statistics were gathered on the connectivity of the different operations in each algorithm of the benchmark set; the total count of the various types of arcs for all benchmarks is shown in Figure 1. Here 1:m denotes an arc in the signal flow graph which connects a single source to m destinations, while n:1 denotes an arc which connects n sources with one destination. The 1:1

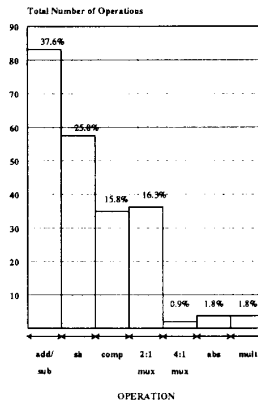


Figure 2: Total Number of Ops vs. Op Type

arcs are dominant for all algorithms; this demonstrates the spatial locality of the computations due to the highly pipelined nature of the algorithms.

We have also generated statistics on the computation rate and the I/O bandwidth requirements for each benchmark (which are not presented here due to space considerations). We can estimate the balance of the required computation rate and the I/O bandwidth of these systems, by tabulating the ratio of these two metrics: the arithmetic mean of the ratio is 6.4 Mops/MB/sec, but a better estimate (to reduce the effect of outliers) is probably the geometric mean which is 3.56 Mops/MB/sec. Clearly, an imbalance of resources would result in a relatively large digression from these values in our final architecture.

We have found the hardware mapping of these algorithms to be supported extremely well by clusters of execution units or EXUs, each containing local register files and connected by a flexible high bandwidth network. This combination of a flexible local interconnect and distributed memory supports the direct mapping of flowgraphs to EXUs, as well as the multiplexing of more than one operation onto a given EXU. In addition, clusters can communicate with other clusters with high bandwidth via 128 dedicated I/O pins. The architecture of our prototype chip is outlined in Figure 3: it contains a cluster of eight EXUs connected by a dynamically configurable crossbar; future members of the family will have two and four times the processing power. We are currently examining two approaches to expansion, one using larger VLSIs and the other using multi-chip modules (MCMs). At the maximum operating frequency of the chip (25 MHz), the Peak Computation Rate to I/O ratio for a thirty-two EXU processor is 800 Mops to 400 MB/sec or 2.0 Mops/MB/sec, a ratio similar to the geometric mean of 3.56 Mops/MB/sec for the benchmark set.

Figure 4 shows how several chips might be applied to form a simple address generation unit (AGU). At power up, the PADDI chips are configured by self-booting from the boot ROM. After receiving a "start" signal from the external controller, the chips are able to perform both data computation and memory addressing tasks. The external controller determines the global instruction sequence and issues the appropriate loads and stores to external memory. Flags generated from the PADDI chips (and the external world) can be monitored by the external controller for global branching.

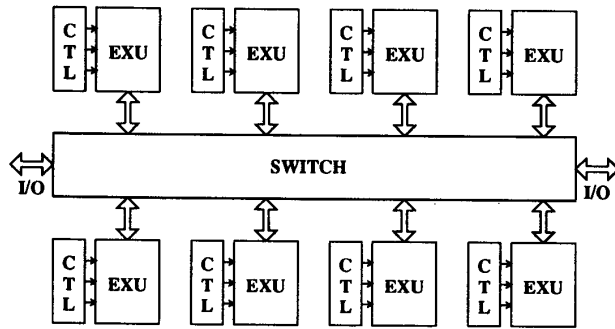


Figure 3: Prototype Architecture

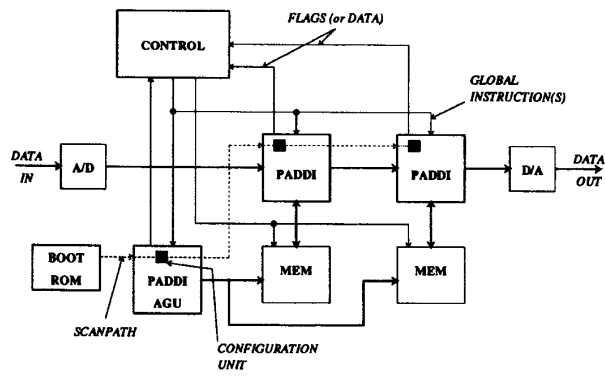


Figure 4: System Using PADDI Chips

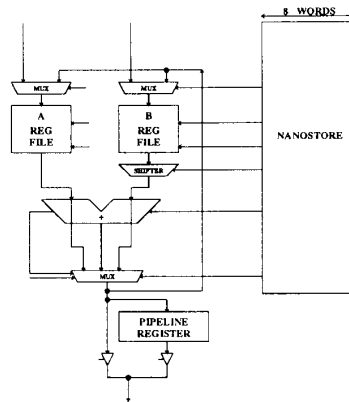


Figure 5: EXU Architecture

## 2.1 Prototype Architecture

**EXU Architecture.** Figure 5 shows the internal architecture of an EXU; it supports addition (either two's complement or unsigned with automatic saturation), subtraction, saturation, comparison, maximum-minimum, and arithmetic right shift. A fast carry-select adder and logarithmic shifter are used to implement the arithmetic functions, and a status flag ( $a \geq b$ ) is available to other EXUs and the external world. Furthermore, the EXUs can be user configured to be both 16 or 32-bit wide.

Two dual-ported register files, each containing six registers, are used for the temporary buffering of data. They can also be configured as delay lines to support the pipelining and retiming of operations. In each file, one of the six registers is configured as a scan-register; in addition to having the functionality of a regular register, this scan-register can be initialized to contain an arbitrary value (for constants) and used for scan-testing. A pipeline register is also available at the output of each EXU for optional use.

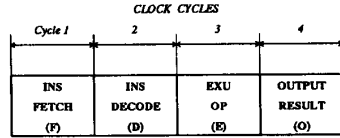
**Communication Network.** To ensure flexible and high bandwidth data routing, a crossbar network has been selected to interconnect the processors; this network routes both data as well as status flags. The data routing is under program control and can be changed in each program cycle, while the routing of the status flags is static and set at compile-time. By making the flag routing static we were able to reduce the required number of nanostore bits and therefore remain within a reasonable silicon budget.

Four input channels and four output channels, all sixteen-bits wide, are also connected to the crossbar network to provide an external data I/O bandwidth of 400 MByte/sec.

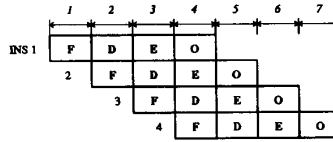
**Control.** Each EXU has an SRAM-based nanostore which is configured serially at set-up time. At run-time, an external sequencer broadcasts a 3-bit global address to each nanostore which is locally decoded into a 53-bit instruction word.

Each EXU operates using a four stage pipeline as shown in Figure 6 (a). Here "Ins Fetch" refers to the fetching of a global instruction from the external controller, while "Output Result" refers to the availability of result at the output pads (e.g., for an external memory write).

The architecture supports two types of branches: *local* and *global*. For local branches, or



(a)



(b)

Figure 6: Four Stage Pipeline

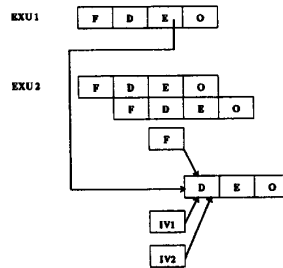


Figure 7: Local Delayed Branches

interrupts, any EXU on the same chip can alter the control flow of any other EXU on the same chip after 1 delay slot, as shown in Figure 7. To receive an interrupt, an EXU must have set one or both of its interrupt enable flags in the previous instruction. Upon receipt of an interrupt, it vectors to one of two pre-compiled instructions (denoted as IVs, for interrupt vectors) in the next cycle. If the EXU being interrupted resides on a different chip, an additional branch delay slot is required. For global branches, an EXU flags the external controller which can then alter the global control flow; in this case, two branch delay slots are required.

**Configuration.** All chip configuration registers are connected as a serial shift register. On board FSMs generate the necessary clock and internal control signals, allowing a chip to boot directly from standard EPROMs.

## 2.2 Architectural Constraints

The *hardware sharing ratio* is defined as the number of different operations that an EXU performs. For example, in a fully pipelined application the ratio is one since the EXUs perform the same

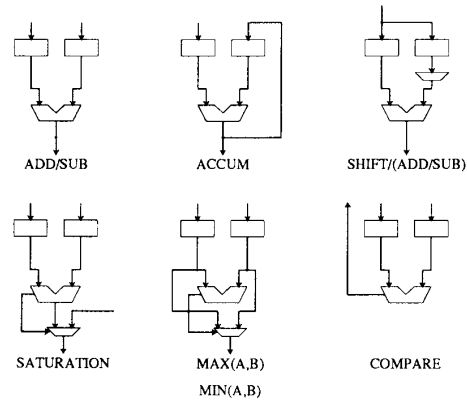


Figure 8: Primitive PADDI Operations (a)

operation every clock cycle. The maximum hardware sharing ratio is constrained by two bounds. The first is the maximum number of registers in the EXUs, which is currently six due to silicon area considerations; the required storage for the operands of the set of operations which get mapped to a particular EXU may not exceed this bound. The second is the nanostore size, which is currently eight words; the maximum number of unique operations that any given EXU can perform is set by this bound. The minimum of these two bounds determines the maximum hardware sharing ratio. At the cost of additional hardware, the number of registers and nanostore size can be varied for other members of the chip-set depending upon the performance range targeted.

### 2.3 Instruction Set Summary

The basic PADDI operations are shown in Figure 8 and Figure 9 respectively. Each EXUs operation in a given cycle is specified by a 53-bit instruction word.

## 3 Low-level Programming Tools

The low-level programming tools, the PAS assembler and PSIM simulator, provide the foundation for the higher-level synthesis tools. PAS represents the lowest software level interface between the programmer and the PADDI architecture, providing a concise method for describing algorithms. The PAS assembly language was designed and implemented with the interconnection network of the PADDI architecture in mind: programs written in it can easily exploit intercommunication between execution units. The intercommunication follows a “receiver controlled” model in which the receiving unit controls the routing of the actual communication while the broadcasting unit only concerns itself with the data or flag to be communicated (except when broadcasting to the external world; in this case the broadcaster must specify which output bus to employ). In addition to being able to express all available PADDI operations in a convenient C-like syntax, the assembler also allows for the explicit specification of instructions within the nanostores at the individual bit level.

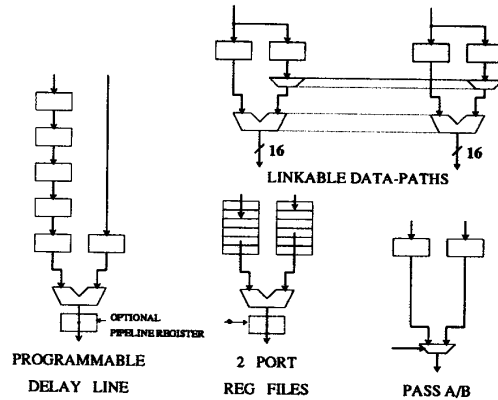


Figure 9: Primitive PADDI Operations (b)

PSIM serves as a tool for simulating and debugging multiple chip PADDI algorithms in software (see Figure 10); the simulation environment supports many of the expected debugging features, including single-stepped execution, breakpoints, and the ability to modify registers and instructions "on the fly."

#### 4 Software Compilation

The major obstacle in using FPGAs and VSPs effectively is the lack of efficient and fast CAD support tools which start from a high abstraction level. Therefore, from the beginning of the PADDI project, special attention has been paid to developing CAD tools which will enable a mapping from a high-level language onto PADDI. CADDI (Compiler for PADDI) is the software environment which we are currently developing and which provides an automated path from the Silage high-level data flow language to the PADDI chip. Its tool suite includes behavioral simulation, estimation, transformations, partitioning, assignment, scheduling, and code generation (see Figure 11), all of which are accessible from within an X-based GUI.

The software environment is being developed as a part of a unified rapid prototyping framework environment which also includes the HYPER [25] and MCDAS [13] systems. The three systems share a common database structure, but each targets a different architecture: CADDI targets field programmable architectures; HYPER, semi-custom architectures; and MCDAS, multiple programmable processor architectures. Besides the common database structure and several tools, CADDI shares with MCDAS and HYPER the same fully modular software organization, making it relatively easy to add new tools or to modify existing tools.

The goal of the CADDI software compilation is to find the best implementation which reflects the speed vs. area balance as specified by the designer. There are two paths of action through the compilation path: one finds the minimum execution time implementation of an algorithm given hardware constraints and the other finds the minimum hardware implementation given a time constraint.

The input to the PADDI software environment (Figure 11) is an algorithm described in the

```

psim
plitvice bigquad 55a pas bigquad e
0 errors, 0 warnings, and 0 messages total
7k bytes allocated
plitvice bigquad 56a psim bigquad e
psim 0.9 (compiled Apr 18 1992)
Reading environment configuration file...
Constructing simulation environment
Loading object file 'bigquad.obj' for chip 'chip_one'
Loading on-line help system
(psim) breakpoint chip_one * 3
breakpoint set at 'chip_one'. exu A. instruction 3
breakpoint set at 'chip_one'. exu B. instruction 3
breakpoint set at 'chip_one'. exu C. instruction 3
breakpoint set at 'chip_one'. exu D. instruction 3
breakpoint set at 'chip_one'. exu E. instruction 3
breakpoint set at 'chip_one'. exu F. instruction 3
breakpoint set at 'chip_one'. exu G. instruction 3
(psim) run
breakpoint at 'chip_one'. exu A. instruction 3
(psim) dumpexu chip_one a pc
chip 'chip_one'. exu A (file 'bigquad.obj')
A1=0 A2=0 A3=0 A4=0 A5=0 A6=-2 B1=0 B2=-2 B3=2 B4=0 B5=0 B6=0
opreg=0 delA=0 delB=0 signed=1 link=0 frciout=00 intvec1=0 intvec2=0
fsw1=0000000000b fsw2=0000000000000000b intarc1=null intarc2=null
result=2 flag=1 lastpc=2 pc=3 int1=0 int2=0
dregh=6 sregh=3 dregb=6 sregb=3 i=0 add1 max=0 hsel=0 hin=0 gsel=0
latch=0 asel=0 bsel=0 ien1=0 ien2=0 frciout=10
sw1=00000001100b sw2=000000000000000100b srcA=XB srcB=XB
(psim) dumpexu chip_one b none
chip 'chip_one'. exu B (file 'bigquad.obj')
A1=0 A2=0 A3=0 A4=0 A5=0 A6=0 B1=0 B2=0 B3=0 B4=0 B5=0 B6=0
opreg=0 delA=0 delB=0 signed=1 link=0 frciout=00 intvec1=0 intvec2=0
fsw1=000000000000b fsw2=0000000000000000b intarc1=null intarc2=null
result=0 flag=1 lastpc=2 pc=3 int1=0 int2=0
dregh=2 sregh=6 dregb=2 sregb=6 r=2 add0 max=0 hsel=0 hin=0 gsel=0
latch=0 asel=0 bsel=1 ien1=0 ien2=0 frciout=00
sw1=00000000100b sw2=000000000000000100b srcA=XB srcB=this_exu
(psim)
    
```

Figure 10: Typical Psim Session

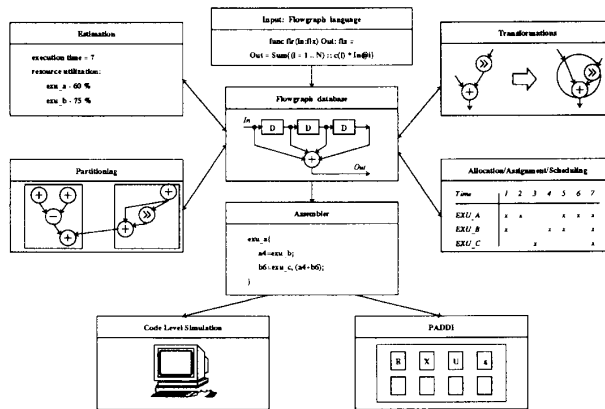


Figure 11: CADDI Software Environment

Silage language [21]. The Silage code is parsed and compiled into a control-flow/dataflow graph (CDFG), which serves as the working structure on which compilation operations are performed and on which results are stored. The nodes represent data operations, and the data edges represent data precedences between nodes; in addition, control edges are introduced to enforce extra precedence rules (for example, when an operation  $A$  has to precede an operation  $B$  by at least  $k$  cycles). The CDFG also supports several macro control flow operations such as loops and if-then-else blocks, the introduction of which results in a hierarchical graph.

The Silage-to-flowgraph translator and the behavioral simulation program are directly adapted from HYPER and MCDAS tools. In addition, several transformations (including retiming, associativity, pipelining, commutativity, and loop unrolling), estimations, resource allocation, scheduling and assignment are built using HYPER tools as their basis.

As a starting point, our tools specifically target the prototype chip containing eight EXUs. New modules for operation chaining and code generation have been developed and partitioning is being refined for multi-chip applications.

#### 4.1 Estimations

Estimations in CADDI include evaluating the critical path, finding the minimum and maximum bounds on PADDI chips given a fixed time allocation, and finding the minimum bounds on the execution time given a fixed allocation of PADDI chips.

Estimations are useful for a variety of reasons during the design and mapping process. Throughout the design process, important decisions which affect the quality of the final solution are often made on an ad hoc basis, and evaluating the effect of a decision requires a time consuming run through the design process; therefore, it would be useful if the designer had additional information to assist in making some of these decisions. In attempting to find a solution that meets an execution time constraint, for example, knowing the critical path can make it easier to evaluate whether a retiming for critical path transformation would be effective. Estimations can also help to give an idea of the maximum improvement that can be expected from a transformation.

The minimum bounds serve as initial solutions and along with the maximum bounds, delimit the search space, thus expediting the compilation process. Estimation of the optimal solution can help determine the absolute quality of a compilation algorithm (such as assignment or scheduling). Since most algorithms are at least NP-complete, benchmarking (even with its associated limitations) is the dominant approach at present.

Techniques to derive sharp lower and upper bounds on hardware for a given time constraint using a discrete relaxation technique have been developed [25]. This relaxation approach turns an NP-complete problem into a polynomial-bound one by relaxing some of the constraints such as precedence relations. While the relaxed minimum hardware bound problem cannot be solved directly, it can be defined as an iterative version of its dual formulation: given an available amount of hardware resources, find the minimum execution time. The optimum solution for the polynomial complexity problem can then be used as a minimum bound for the initial NP-complete problem. We have modified these techniques to address the PADDI architecture: by classifying the nodes in the CDFG as either *exu* type, *input* type, or *output* type, the required minimum and maximum hardware bounds on the EXUs, input channels and output channels can be estimated using the above technique. Any mapping should therefore not violate the hardware constraints, which for one prototype chip are eight EXUs, four input and four output channels. For EXU, input, and output bounds, the corresponding number of PADDI chips which would be necessary to satisfy those bounds are also computed and displayed.

The second type of estimation in CADDI determines the minimum execution time given a set number of available resources. Solving this problem does not incur additional computation since this information is produced as a by-product when computing the hardware bounds through the dual discrete relaxation problem formulation.

## 4.2 Transformations

Transformations change the CDFG structure to improve some aspect of the final implementation while preserving the input/output relationship; they also have the potential to drastically alter constraints such as the critical path through the CDFG. Thus, transformations are a key ingredient in both software and high-level synthesis compilers, and CADDI gives special attention to them. In CADDI, transformations are classified as either architecture independent or architecture dependent transformations: both types assume a basic hardware model, but the dependent ones are explicitly tailored to the PADDI data path and structure. Architecture independent transformations [24] developed in the HYPER system have been adopted and can be applied in CADDI. These include pipelining, retiming, algebraic transformations (e.g., associativity, distributivity, and commutativity), and loop unrolling.

In addition to architecture independent transformations, architecture dependent transformations, such as operation chaining have been developed. Module selection and operation chaining have recently received much attention in high-level synthesis [10, 7]. In our case, since the PADDI architecture is fixed, no module selection is involved. Operation chaining, however, plays a significant role: by exploiting the structure of PADDI's data path, chaining reduces execution time by collapsing two operations that would otherwise take two control cycles into a single execution step. Operation chaining in the CADDI environment has a rather distinctive nature compared to the traditional chaining process. First, since the hardware platform is fixed, there is no need for timing analysis, which is often the most difficult and least reliable part of high-level synthesis operation chaining and module selection. Second, the process is restricted to the chained operations which are supported by the PADDI EXU: shift/add and shift/subtract. Note that the algorithm can easily be modified for use with different data paths where alternate operations are available for chaining; in fact, the techniques developed here are an instance of a more generic chaining problem that will arise in compilers for special purpose architectures.

A series of operation chaining transformations are performed to transform the CDFG nodes to match PADDI primitive operators. Shift/add and shift/subtract conversions can result in a significant reduction in of the number of nodes that need to be scheduled, especially in fixed coefficient multiplication applications.

Since the order in which transformations are applied highly influences their effectiveness, we are currently using two different approaches for this task. In the first case, a fixed set of scripts which invokes transformations in a particular order is developed using experience from a set of benchmarks. Another option is the user guided application: in this case, after each step (the application of a particular transformation), estimation is performed and the user is informed of the effects of the transformation. A record of steps and all intermediate results are maintained, so the user can easily backtrack if he or she changes his or her opinion about the effectiveness of the current scenario.

## 4.3 Partitioning, Allocation, Assignment, and Scheduling

The Data Path Partitioning Module, of which a first prototype has been developed, clusters operations onto the available chips in the cases when more than one PADDI chip is used. The goal

is twofold: to minimize inter-chip communication and to maximize resource utilization of the PADDI chips.

The Allocation Module allocates a number of chips and the available number of clock cycles. If we are minimizing hardware for a fixed time, it successively increases hardware until a successful schedule is found. If we are instead minimizing time for a fixed hardware, it successively increases the time allocation until a successful schedule can be obtained.

The Assignment Module maps operations to particular EXUs and determines the registers from which to obtain and store data, while the Scheduling Module assigns operations to given control steps. These tasks are clearly interdependent. Each is itself an NP-complete problem, and together they form an even more difficult problem, so heuristics must be applied.

Because we are mapping to fixed hardware, we have chosen to first perform resource allocation, then assignment, and finally scheduling. A great number of high-level synthesis systems perform these tasks in just the opposite order, where flexibility exists to add hardware resources as needed.

It was of key importance to select an approach that satisfies the following requirements. First, the model must accurately represent all three components of hardware cost (execution units, memory, and interconnect). This is imperative in PADDI because of the hardware constraints. Second, useful feedback must be made available to the compilation framework and/or to the user. This feedback could include information on eventual bottlenecks in the algorithm, such as insufficient time allotment, lack of parallelism, or underutilization of hardware. This information can be used to decide which graph transformations (such as more pipelining) to apply, or which hardware unit to add. Third, the previously described ordering (allocation, assignment, then scheduling) must be met. Based on this analysis, we have opted for an approach described in [23].

The allocation starts with the minimum bounds as computed in estimation. Assignment is done in a probabilistic fashion, and is based on the rejectionless antivoter assignment algorithm [23]. The assignment has several constraints built in to address the PADDI hardware platform, whose model for one prototype chip includes 8 general purpose EXUs, 4 input units, and 4 output units. The selection of a given assignment is based on a calculated quality measure of the chance of finding a corresponding successful schedule. Once an assignment is accepted, scheduling is performed for speed optimization using resource utilization as the priority function. Because of hardware constraints inherent to PADDI, we give higher scheduling priority to operations that will relax constraints on these critical resources.

#### 4.4 Mapping to PADDI

The final step in the compilation process is the mapping of the CDFG onto the PADDI architecture, a step analogous to the hardware mapping task in high-level synthesis. The translator from the CDFG to the PAS assembly language uses the assignment and scheduling information to generate instructions for each of the PADDI EXUs. This step is much more straightforward, however, than hardware mapping, since all components are fixed and what is mainly required is a direct translation. Currently the static configuration settings for the chip are modified by the user as necessary: in the future, these will be set automatically.

Figure 12 presents a typical compilation session as might be seen by a user. Note that the path may involve several iterations of estimation, partitioning, assignment, and scheduling to view the effect of various transformations.

#### 4.5 Results

This section presents several examples taken through various compilation steps.

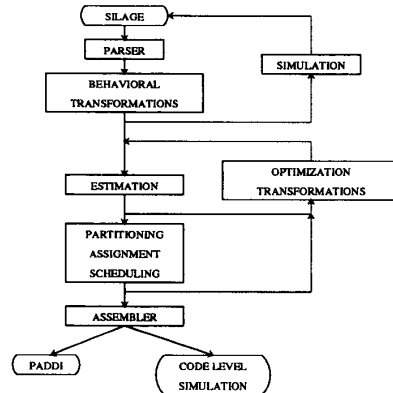


Figure 12: Typical CADDI Compilation Session

Example	before chaining	after chaining	percent reduction	available time
biquad	3/4/5	2/4/4	33/0/20	5
6th order BP IIR	4/9/15	3/8/13	25/11/13	15
7th order HP IIR	5/11/20	3/11/17	40/0/15	20
11 tap FIR	4/18/11	3/14/11	25/2/0	11
21 tap Hamm LP FIR	19/20/8	10/20/7	47/0/13	8

Table 2: Operation Chaining (Min EXUs/Max EXUs/critical path)

Table 2 shows the estimation results on minimum EXUs, maximum EXUs, and the critical path before and after the chaining of shift/add and shift/subtract on several fixed coefficient filter examples. We fixed the available time to be the initial critical path and performed estimations on the examples before and after operation chaining. In most cases, we observe significant reduction in hardware bounds and critical path due to the operation chaining.

After operation chaining, we then applied a series of transformations to each example in order to increase the throughput. Before transformations, the available time was set to the critical path and the maximum and minimum hardware bounds were found. After transformations were applied, the available time was set to the new critical path and the new maximum and minimum hardware bounds were established. The results are shown in Table 3: note the throughput improvement in all benchmarks. Greater improvement is seen in the FIR filters rather than the recursive ones due to the inherent iteration period bound of the latter filters. Future transformations such as those described in [29, 22] may result in further improvements for these cases.

The next step involves actual mappings to PADDI using allocation, assignment, and scheduling. Starting from the results of Table 2, for a fixed available time, we applied transformations for resource utilization, and then applied allocation, assignment, and scheduling. In most examples, we set the available cycles to 8 and performed scheduling for resource utilization. For the biquad, however, increasing the number of cycles past 5 does not result in further improvements. Recall that in order for an application to be mapped onto one prototype chip, the number of unique

Example	before transformations	after transformations
biquad	2/4/4	4/5/3
6th order BP IIR	3/8/13	7/13/7
7th order HP IIR	4/11/17	16/18/2
11 tap FIR	3/14/11	38/38/1
21 tap Hamm LP FIR	19/20/7	50/50/1

Table 3: Critical Path Reduction Using Transformations (Min EXUs/Max EXUs/critical path)

Example	Min and Max bounds	Actual EXUs used	available time	EXU utilization (%)
biquad	2/5	2	5	90
6th order BP IIR	5/14	8	8	62
7th order HP IIR	4/18	4	8	94
11 tap FIR	4/17	5	8	80

Table 4: Comparison of Estimation and Scheduling results

instructions in the program instruction list must be no more than eight, and the number of EXUs used must be at most eight. If more EXUs are necessary, we must partition the algorithm and map it onto several PADDI chips. Except for the Hamming low-pass FIR, all examples were mapped onto a single prototype chip.

Table 4 shows the actual hardware used vs. the hardware minimum and maximum bounds produced by estimations. The last column shows the resulting schedule's percent utilization of allocated hardware. Clearly, for these examples, area utilization is high, and the actual mappings compare excellently to the estimation bounds.

## 5 Conclusion

The architecture, low-level programming tools, and CAD framework of an integrated system for the rapid prototyping of real-time data paths has been described; this system is targeted towards high performance digital signal processing applications. A prototype chip containing eight EXUs was fabricated and functions at 25 MHz. A 16 EXU (400 MIPS) processor is currently under design, together with a multi-chip module approach which can support up to 32 EXUs (800 MIPS) in a single package. An assembler and simulator which form the core of the low-level programming tools were described.

The estimation, transformation, assignment, and scheduling portions of the compilation system have been presented. The quality of the presented algorithms on the PADDI hardware platform has been demonstrated by comparing the results to the estimated minimum bounds on several examples. The initial results obtained are promising, and the next step is to implement automatic partitioning and to enhance the compilation steps to exploit more fully all the functionality available in PADDI.

## REFERENCES

- [1] J. Bier, S. Sriram, and E. Lee. "A Class of Multiprocessor Architectures for Real-Time DSP". *VLSI Signal Processing IV*, pages 295-304, Nov. 1990.
- [2] R. Brodersen and J. Rabaey. "Evolution of Microsystem Design". In *ESSCIRC'89: Proceedings of the 15th European Solid State Circuits Conference*, pages 208-217, Sept. 1989.

- [3] D. Chen. "Programmable Arithmetic Devices for High Speed Digital Signal Processing". PhD thesis, UC Berkeley, May 1992.
- [4] D. Chen, L. Guerra, E. Ng, D. Schultz, C. Yu, and J. Rabaey. "A Field Programmable Architecture for High Speed Digital Signal Processing Applications". presented at the 1992 ACM International Workshop on Field- Programmable Gate Arrays, pages 117-122, Feb. 1992.
- [5] D. Chen and J. Rabaey. "PADDI: Programmable Arithmetic Devices For Digital Signal Processing". "VLSI Signal Processing IV", pages 240-249, Nov. 1990.
- [6] D. Chen and J. Rabaey. "A Reconfigurable Multiprocessor IC for Rapid Prototyping of Real Time Data Paths". In *Proceedings International Solid State Circuit Conference*, pages 74-75, Feb. 1992.
- [7] C. Chu and J. M. Rabaey. "Hardware Mapping and Module Selection in the HYPER Synthesis System". *Proc. European Conference on Design Automation*, pages 176-180, 1992.
- [8] P. Duncan, S. Swamy, S. Sprouse, D. Potasz, R. Jain, N. Gafter, W. Cammack, Y. Wang, and W. Gass. "Hi-PASS: A Computer-Aided Synthesis System for Maximally Parallel Digital Signal Processing ASICs". In *Proc. ICASSP 92: 1992 International Conference on Acoustics Speech and Signal Processing*, pages V605-V608, Mar. 1992.
- [9] M. Engels, R. Lauwereins, and J. Peperstraete. "Rapid Prototyping for DSP Systems with Multiprocessors". *IEEE Design & Test of Computers*, pages 52-62, June 1991.
- [10] R. Jain et al. "Module Selection for Pipelined Synthesis". In *Proceedings 25th ACM/IEEE Design Automation Conference*, pages 542-547, 1988.
- [11] R. Freeman. "User-programmable Gate Arrays". *IEEE Spectrum*, pages 32-35, December 1988.
- [12] T. Fukushima. "A Survey of Image Processing LSIs in Japan". In *IEEE International Conference on Pattern Recognition*, volume 2, pages 394-401, 1990.
- [13] P. Hoang and J. Rabaey. "McDAS: A Compiler for Multiprocessor DSP Implementation". In *Proc. ICASSP 92: 1992 International Conference on Acoustics Speech and Signal Processing*, pages V581-V584, Mar. 1992.
- [14] J. Rabaey and R. Brodersen and A. Stölzle and S. Narayanaswamy and D. Chen and R. Yu and P. Schnupp and H. Murveit and A. Santos. "A Large Vocabulary Real Time Continuous Speech Recognition System". *VLSI Signal Processing III*, pages 61-74, 1988.
- [15] R. Kavalier. "The Design And Evaluation Of A Speech Workstation". Technical Report Memo. No. UCB/ERL M86/39, U.C. Berkeley, 1986.
- [16] K. Kitagaki, T. Oto, T. Demura, Y. Araki, and T. Takada. "A New Address Generation Unit Architecture for Video Signal Processing". In *Proc. of Visual Communications and Image Processing*, pages 1-10, Nov. 1991.
- [17] S. Kung. "VLSI Array Processors". Prentice Hall, 1988.
- [18] P. Lippens, J. van Meerbergen, A. van der Werf, W. Verhaegh, B. McSweeney, J. Huisken, and O. McArdle. "PHIDEO: A Silicon Compiler for High Speed Algorithms". *Proc. European Conference on Design Automation*, pages 436-441, Feb. 1991.
- [19] S. Note, W. Geurts, F. Cathoor, and H. D. Man. "Cathedral III: Architecture-Driven High-level Synthesis for High Throughput DSP Applications". *28th ACM/IEEE Design Automation Conference*, June 1991.
- [20] S. Note, J. V. Meerbergen, F. Cathoor, and H. D. Man. "Hardwired Data Path Synthesis For High Speed DSP Systems With The Cathedral III Compilation Environment". In *Logic and Architecture Synthesis for Silicon Compilers*, pages 243-254. Elsevier Science Publishers B.V. (North-Holland), Feb. 1989.
- [21] P. Hilfinger. "A High Level Language and Silicon Compiler for Digital Signal Processing". In *Proc. IEEE Custom Integrated Circuits Conference*, pages 240-243. IEEE, May 1985.
- [22] K. Parhi and D. Messerschmitt. "Pipeline Interleaving and Parallelism in Recursive Digital Filters, I and II". *IEEE Transactions on Speech and Signal Processing*, pages 1099-1134, July 1989.
- [23] M. Potkonjak and J. Rabaey. "A Scheduling and Resource Allocation Algorithm for Hierarchical Signal Flow Graphs". In *Proceedings 26th ACM/IEEE Design Automation Conference*, pages 7-12, June 1989.
- [24] M. Potkonjak and J. Rabaey. "Optimizing Resource Utilization Using Transformations". In *IEEE International Conference on Computer-Aided Design*, pages 88-91, Nov. 1989.
- [25] J. Rabaey, C. Chu, P. Hoang, and M. Potkonjak. "Fast Prototyping of Datapath-Intensive Architectures". *IEEE Design & Test of Computers*, pages 40-51, June 1991.
- [26] P. Ruetz and R. Brodersen. "A Realtime Image Processing Chip Set". In *Proceedings International Solid State Circuit Conference*, pages 148-149, Feb. 1986.
- [27] R. Schmidt. "A Memory Control Chip for Formatting Data into Blocks Suitable for Video Coding Applications". *IEEE Transactions on Circuits and Systems*, pages 249-258, Oct. 1989.
- [28] U. Schmidt. "Data Wave - a Data Driven Video Signal Array Processor". In *Hot Chips II : A Symposium on High Performance Chips*, Aug. 1990.
- [29] M. A. Soderstrand and B. Sinha. "Comparison of Three New Techniques For Pipelining IIR Digital Filters". In *Asilomar Conference on Circuits and Systems*, pages 439-443, 1985.
- [30] J. Turner. "Almost All k-Colorable Graphs are Easy to Color". *Journal of Algorithms*, 9(1):63-82, 1988.
- [31] A. van Roermund, P. Snijder, H. Dijkstra, C. Hemeryck, C. Huzier, J. Schmitz, and R. Sluiter. "A General Purpose Programmable Video Signal Processor". *IEEE Transactions on Consumer Electronics*, pages 249-258, August 1989.