Chapter 7 : Execution pipeline

The execution unit contains the major data processing logic of the processor. It provides an autonomous multiplier, a barrel shifter connected to one of the operand buses, an ALU and storage for the Current Processor Status Register (CPSR). The interconnection of these elements is shown below in figure 7-1 (the shaded boxes represent pipeline latches).

When the multiplier is not required, there is an internal bypass mechanism which passes its inputs straight through to the outputs.

7.1 The multiplier

The multiplier in AMULET1 is substantially different from the '2-bits at a time' Booth's multiplier used in the ARM6 [Furb89, Page 253]. A multiplication using the ARM6 involves a complete cycle around the datapath for every 2 bits of the multiplicand with the intermediate results being stored temporarily in the register bank.

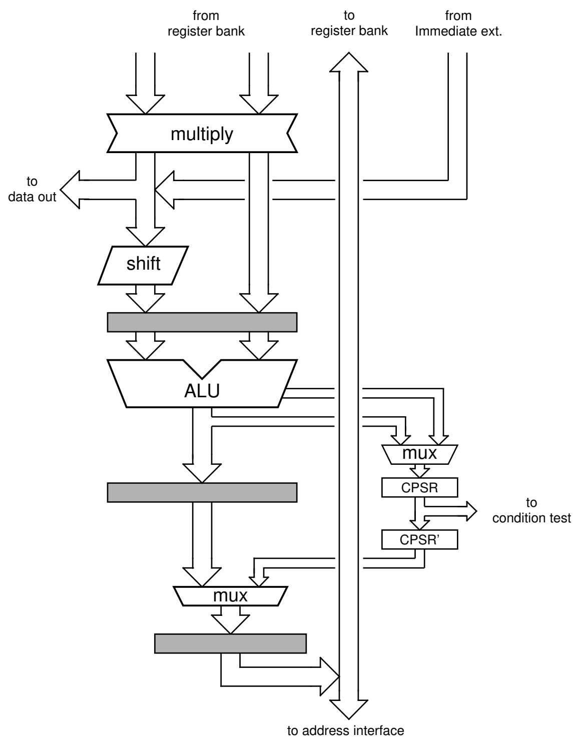


Figure 7-1: Execute pipeline

Multiplication in AMULET1 is based upon a shift-and-add multiplier using carry-save adders [Day92]. It is an autonomous unit which accepts two source operands and produces the partial product and carries which are added together in the ALU to complete the operation. Internally the multiplier controls the cycles of the shift and add operations and can terminate early when the multiplication is complete.

Each cycle of the multiplication involves:

1. Latching the inputs to the adders.
2. Performing the shift and add operation.
3. Latching the output of the adders

This continues until the multiplication is complete. The speed of operation of the circuit is governed by the speed at which the two-phase control circuitry can open and close the input and output latches. Early designs indicated that the control cycle time was much greater than that required to perform a single 1-bit shift and add operation. Investigations revealed that it was possible to do a 3-bit shift and add operation in the control cycle time so although the cycle time of the multiplier could not be improved, the amount of work done per cycle was increased and the overall performance of the multiplier improved.

As the AMULET1 multiplier is an autonomous self contained unit, it does not activate the entire datapath for every cycle of its operation (thus saving power).

7.2 The shifter

The shifter in AMULET1 is the ARM6 barrel shifter with an added matched path for self-timing purposes. The barrel shifter is constructed as a 32 x 32 cross switch matrix of n-type pass transistors. The circuit is dynamic in operation with the outputs being precharged High before a shift is evaluated. As specified by the ARM architecture, the shifter is only connected to one operand bus and is in series with the ALU.

7.3 The ALU

ARM ALU operations fall into one of three main categories: moving data from input to output, performing logic functions (XOR, AND, OR) and arithmetic operations (addition). Input buffers allow optional zeroing or complementing of operands to support subtraction (using twos-complement).

In addition to the use of the ALU for the sixteen explicit data processing instructions specified by the ARM architecture, the ALU is used implicitly by the other instruction classes (see section 4.3), for example, to calculate the address of a data transfer given the base address and an offset.

7.3.1 Performance considerations

In a synchronous system, the overall performance of the ALU is usually limited by the arithmetic logic. The speed of the addition operation is related to how quickly the carry signals can propagate across the word. The worst case occurs when the carry propagates across all bits in the word. In a synchronous system the clock period is chosen to allow time for this worst case operation although typically the result will be ready much sooner. To reduce the time for the worst case (and hence reduce the clock period) synchronous systems use schemes such as carry look-ahead or carry select [Eshr89].

Without the constraint of an external clock, an asynchronous ALU can be designed to be quick for “typical” operands and slower for worse case operands; there is no need to ensure that the worst case is equally fast, if it does not happen very often. The emphasis, with asynchronous logic design techniques, is to make the average case operation fast but to accommodate the worst case by allowing more time for the calculation. Synchronous systems cannot allow more time for the worst case because the clock period is fixed.

Optimizing the ALU design for the fast addition of “typical” operands only has benefits if the worst case is statistically rare and much worse than the average. Figure 7-2 (a) shows the mean carry propagation distance as a function of the width of the word when the data are random (figures 7-2(a) & (b) are reproduced with the kind permission of Dr. J.D. Garside [Gars93]). It can be noted that the mean carry propagation distance for a 32-bit addition is only 4.4 bits. This is much less than the worst case (32-bit).

In practice, data are not random, so to obtain a more accurate reflection of the mean carry distance a carry length analysis of a dynamic instruction trace was undertaken. The distribution of carry propagation lengths while running a benchmark program is shown in figure 7-2(b). The statistics are divided into address calculations and data operations where the average propagation distance for data operations was found to be approximately 18 bits, whereas the propagation distance for address calculations was found to be only 9 bits. Overall this gave a combined average carry propagation distance of approximately 12-13 bits. This is described in more detail elsewhere [Gars93].

The net result of the small average propagation distance during typical ALU operation is that a simple ripple-carry design can be constructed which, on average, performs better than more elaborate carry lookahead/select adders; but if the (rare) worst case is encountered it takes longer to calculate the answer.

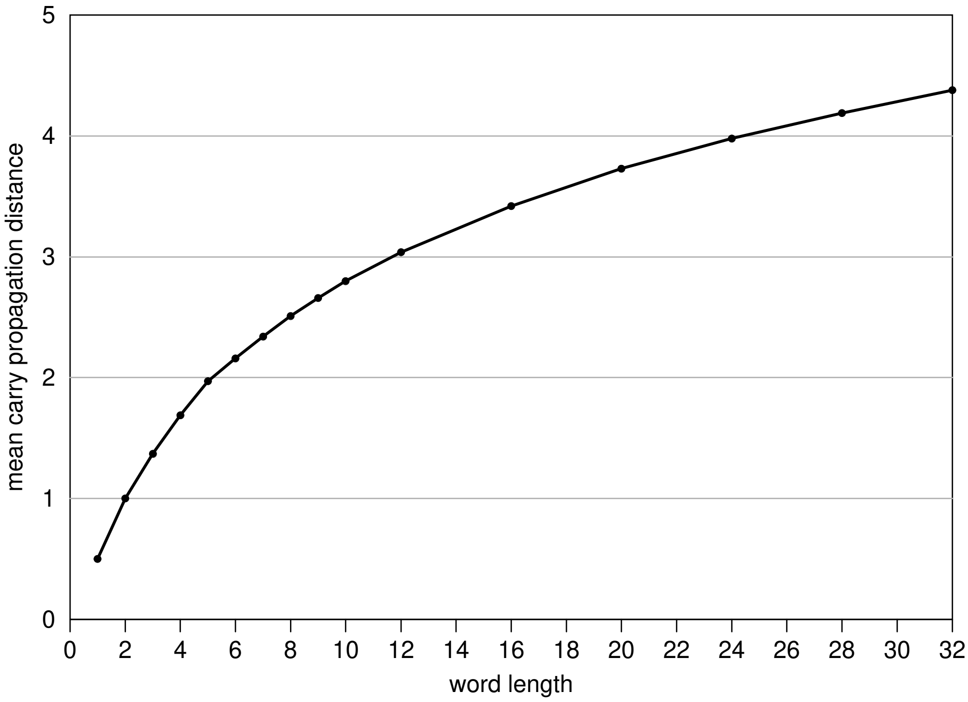
7.3.2 Implementation

The ALU adder in AMULET1 consists of thirty two full adders with no special acceleration logic to speed up carry propagation. The carry signal is encoded in dual-rail format and a completion detection circuit signals to the environment when the carry propagation is complete. The detail transistor circuitry to achieve this is described elsewhere [Gars93].

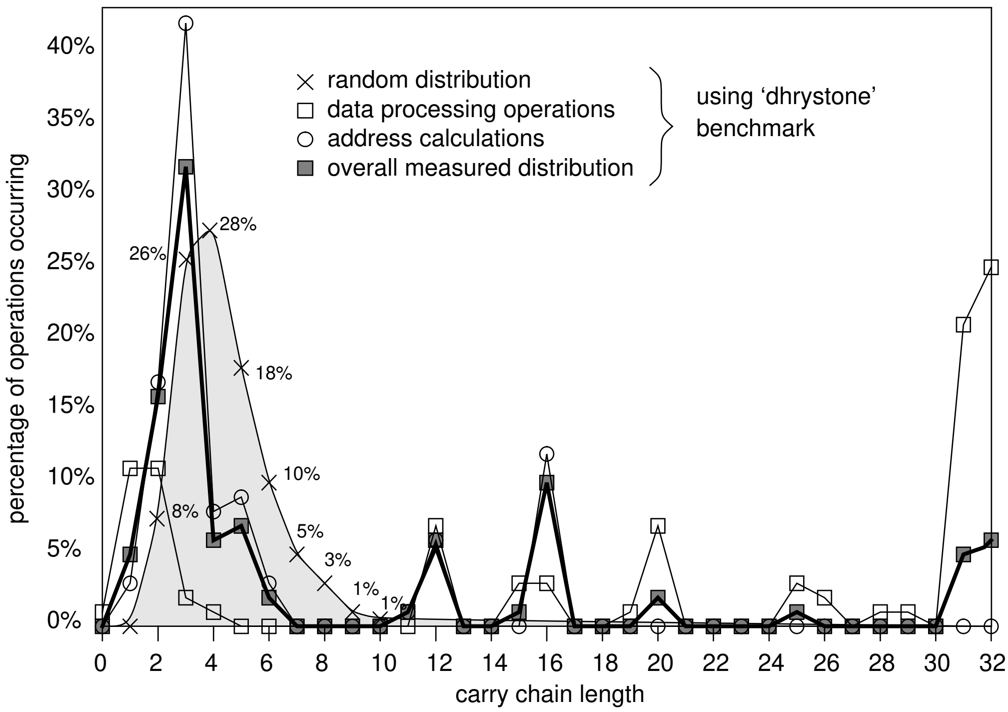
Figure 7-3 shows the resulting silicon layout of the ALU in AMULET1 in comparison to the ALU of the ARM6 drawn at the same scale. The asynchronous ALU of AMULET1 is approximately 40% of the area of its synchronous counterpart. The sparse, area-inefficient, carry-select logic can clearly be seen to the left of the ARM6 ALU.

7.4 The CPSR

The placement of the CPSR was one of the most taxing decisions taken during the design of AMULET1, it contains the arithmetic flags and the processor mode information. The arithmetic flags are generated and used in the ALU but are also made available to the condition evaluation hardware. The control flags are changed either as a result of decode information or by being directly loaded (using an ARM *MRS/MSR* instruction). The mode

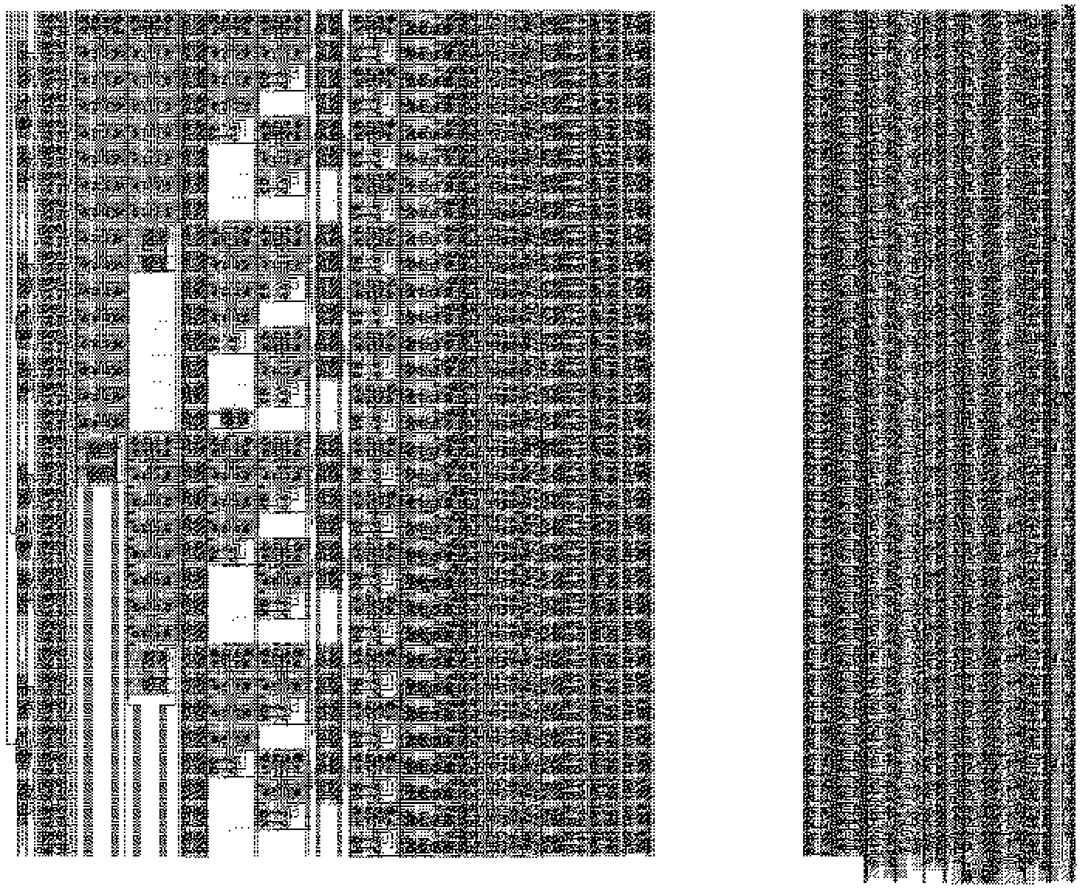


(a) Average size of longest carry propagation for random data distribution



(b) Longest carry chain length distribution - 32-bit word

Figure 7-2: ALU carry propagation statistics



(a) ARM6 ALU (b) AMULET1 ALU

Figure 7-3: ARM6 v AMULET1 ALU area

information is used in the primary decode to determine which subset of the register bank is currently available to the programmer and in the memory interface to ensure that the correct memory privileges are enforced.

The ALU decode is the earliest point at which it is known whether an instruction will execute (i.e. whether it has passed its condition test). It is also the earliest point at which the processor state can be changed, so it is a convenient place to store the processor state. Placing the CPSR close to the ALU is also advantageous because of the close connection required between the CPSR arithmetic flags and the ALU.

An instruction which changes the CPSR mode flags also changes the visible register set (see figure B-4 for the register bank organization). The mode information is directly connected from the CPSR to the register decode circuitry, so that the correct register selection is performed. To ensure correct operation, an instruction which follows a mode changing operation in the pipeline is forced to wait at the primary decode until the potential mode change has taken place. The ALU decode (decode 3) signals to the primary decode when it is safe to continue (i.e. the mode flags are stable and will select the correct register subset).

The CPSR structure consists of 2 latches connected as shown below in figure 7-4. The active CPSR is stored in latch *A* with the output connected to the condition evaluation hardware and the relevant register decode logic. The second latch holds a copy of the CPSR while new values are being calculated. In particular, it holds the previous arithmetic flag inputs to the ALU stable while the new flags are being calculated.

The second latch also has an important role in preserving the original CPSR during exception entry. There are three cycles associated with exception entry; generate and issue the exception vector, copy the CPSR to the SPSR (of the mode being entered) and finally copy the PC into the link register of the destination mode (see section 4.3.7).

During the first cycle, the mode of the processor is changed in latch *A* at the same time as the exception vector is sent to the address interface. The original CPSR is still preserved in latch *B*. The second cycle of exception entry waits in the primary decode for the mode change to take place and then, after locking the SPSR in the new mode, it progresses to the ALU stage where the original CPSR is copied from latch *B* and sent to the SPSR of the new mode in the register bank.

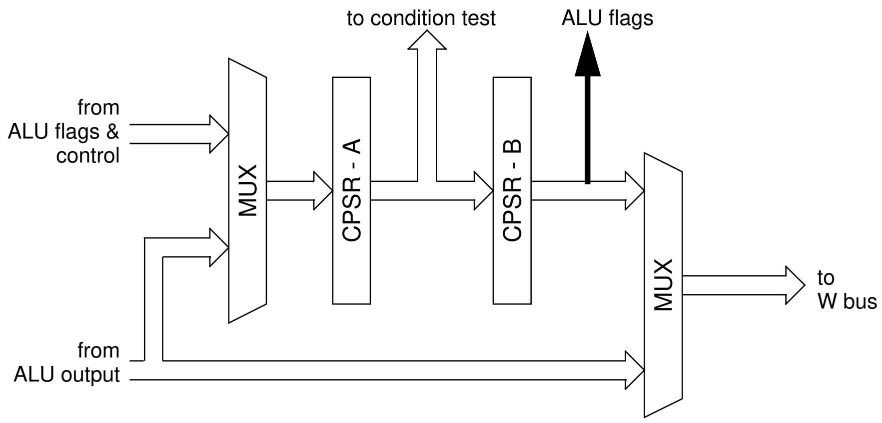


Figure 7-4: CPSR structure

Chapter 8 : Implementation

To implement a complex circuit, such as the asynchronous ARM, an ordered approach to design and verification is required. The design flow used during the construction of AMULET1 has many similarities with that used by ARM Ltd. in the design of the synchronous ARM processors [Furb89, Page 285].

8.1 Design flow

The overall design flow for AMULET1 is shown in figure 8-1. The design process begins by constructing a high-level model. Initially, *Mainsail* [XIDA87, Comp91b], the language used in the Compass Design Automation tools, was used but proved to be inefficient in this application and the resulting simulations took too long to run. Instead, ASIM [Smit92], an internal ARM Ltd. tool was used to model the processor. ASIM provides an efficient event driven simulator and a hardware description language with a library of standard parts. Extra models written in C [Kern88] can be added to the standard library.

A complete design of the processor was developed within the ASIM environment. Verification was performed by connecting the asynchronous processor to a simple simulated memory system and then loading and running the ARM validation programs. In total, over 4 million instruction cycles were simulated and it was during this time that the deadlock situations described earlier were detected (e.g. see chapter 6).

Once a stable processor design was available, this was transferred into schematic form in the Compass Design Automation environment. With detail schematics complete, it is possible to begin designing the silicon geometry. The silicon layout uses efficient custom designed cells for the datapath and standard cells from a cell library for the majority of the control logic. When the layout of a block is complete, it is compared to the schematic using automatic comparison tools (net compare [Comp91c]) to ensure it is correct.

8.1.1 Verification of the design

It is possible to generate static test vectors for a block which adheres to a Micropipeline interface. This is done by placing large delays in all event wires which cross the test interface in the high-level model. The output events of a block indicate that the output data is valid, so a large delay in the request path means that the output data is stable for longer and can be extracted as a test vector. A similar argument can be applied to input bundles with large delays in the event path.

ASIM allows any level of hierarchy to be defined as a test interface for vector generation. This allows the complete processor core to be simulated and the vectors to be generated just for the required interface (note that all event wires crossing the interface must have had the

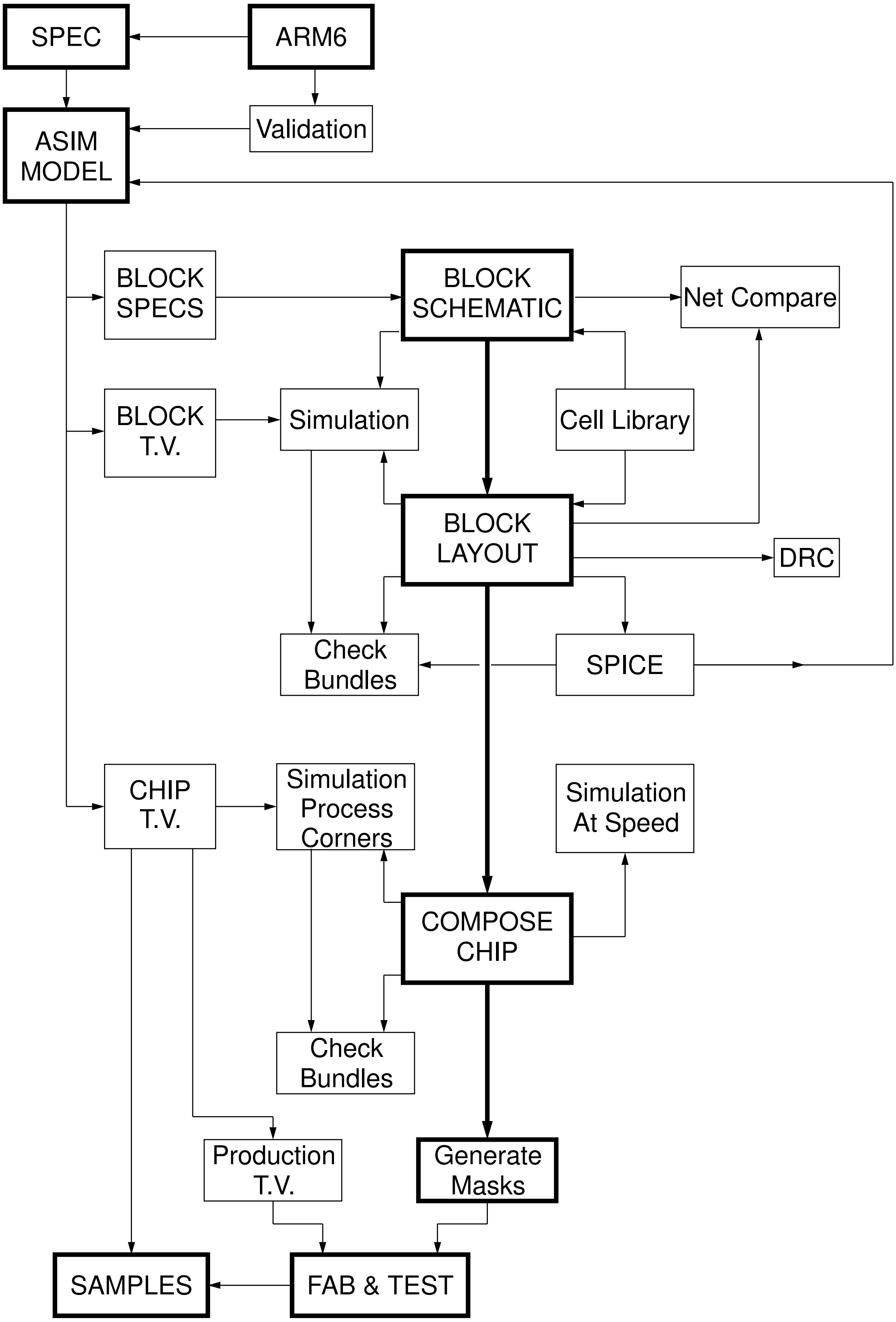


Figure 8-1: Design flow

delays inserted first). Once the vectors have been extracted, ASIM provides a facility to verify that the vectors are static by re-applying them directly to the block to be tested and observing the outputs. Experience has shown that it is relatively straightforward to generate static vectors for a clean Micropipeline interface.

The block level test vectors are translated into the Compass Design Automation environment and can be used to verify the block level schematics and subsequently the extracted silicon layout.

An important consideration of a Micropipeline system is to ensure that all interfaces obey the bundle data convention. This constrains the output data to be valid before an output request is issued and the data must remain stable until an acknowledgement is received (see figure 3-2 in chapter 3).

The bundle constraints are checked at two levels, first by using SPICE to examine any matched paths or self-timing and then, at a higher level, a simple tool was written to check the simulator output. Given a definition of data bundles and their corresponding control, it is relatively straightforward to check that the data was stable before a transition on the request wire and that it did not change until after a transition on the acknowledge wire.

The same verification approach was applied to the complete chip after the blocks of layout were composed and the top level wiring connected. In addition, all test vector simulations were run at all four "process corners" of the silicon (i.e. all combinations of fast and slow n- and p-transistors). This can give a effective speed ratio of n- to p-transistors of 4:1 in both directions.

To allow the design to be verified "at speed" (rather than using just static vectors), the extracted layout of the complete chip was connected to a simulated system. The memory of the system was then loaded with a program and the processor allowed to execute it at its own speed. This "at speed" testing applies only to the particular point in the possible process space to which the simulator is calibrated; to increase test confidence, the program was executed at all four simulated process corners (again using a 4:1 spread in transistor speeds).

The complete chip has a Micropipeline interface, so static test vectors can be generated as described before. Programs carefully chosen to toggle a high percentage of internal nodes can be used to generate the static production test vectors required to verify the chip after fabrication. The static nature of the vectors allows conventional testers to be used.

Much work still needs to be done to address the problem of test and verification of an asynchronous processor (fault simulation etc.). As many parts of the asynchronous datapath are essentially the same as their synchronous counterparts, some of the synchronous test techniques may be applied to them. However, techniques to verify the control need to be further developed.

8.2 Complete organization

The overall complexity of the resulting design is shown in figure 8-2. The diagram features details not previously illustrated:

- The memory control FIFO (*mem ctrl. FIFO*) is connected from the address control information latch to the destination control (*dest. ctrl.*) in the data interface to

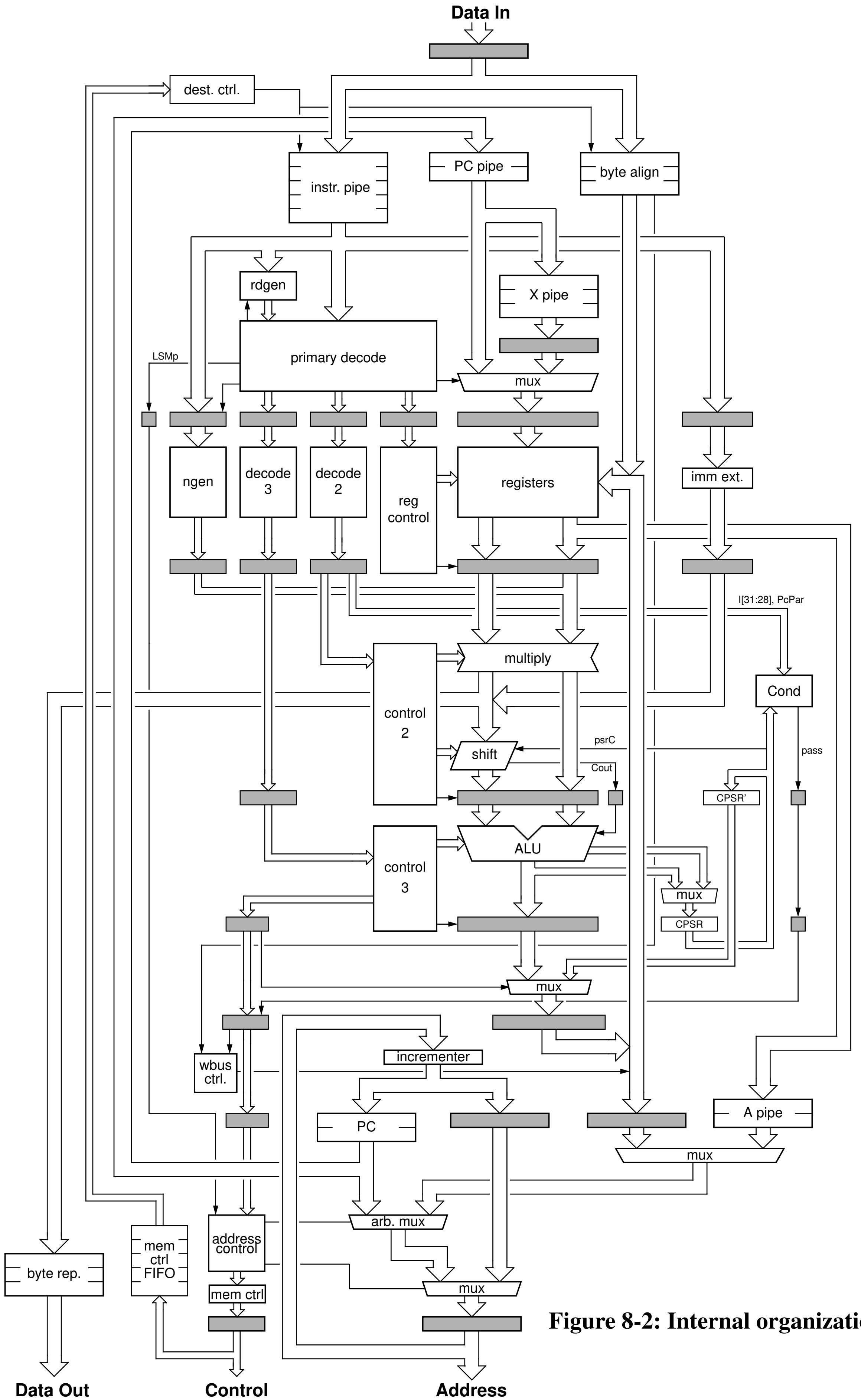


Figure 8-2: Internal organization

allow the incoming memory values to be steered to the correct destination (either the instruction pipeline (*instr. pipe*) or the data-in processing logic (*byte align*)).

- The primary decode and the address interface are connected via the *LSMp* pipeline to facilitate the load/store multiple instruction (see section 4.3.6 & section 6.1.1).
- *Rdgen* which sequentially generates the addresses of the registers to transfer in a load/store multiple instruction given the sixteen bit field from the instruction word (1 bit is set for each register to transfer).
- *Ngen* calculates the number of registers to be transferred in a load/store multiple instruction from the bottom sixteen bits of the instruction. The resulting value is used in base calculations (see section 4.3.6). It is also convenient for *ngen* to generate the vector addresses for exception entry (see section 4.3.7) because of its connection to the datapath *A* bus.
- The write bus control logic (*wbus ctrl.*) which arbitrates requests from either the ALU result register or incoming memory data wishing to use the write bus.

8.2.1 Datapath VLSI organization

Figure 8-3 shows an implementation orientated view of the overall organization, with particular emphasis on the VLSI floorplan of the datapath. The major data buses are shown in blue and the major control dependencies are shown in red. The actual order of the datapath blocks in the diagram are as implemented in silicon. The diagram also shows how some of the multiplexers, shown in figure 8-2, are implemented as a shared bus with each possible source having tri-state drivers onto the bus. For example, the ALU output latch and the CPSR in the previous diagram are shown as being combined with a multiplexer into the result latch. The actual implementation shows a shared bus arrangement.

8.3 Silicon layout

The complete silicon layout of AMULET1 is shown in figure 8-4. This is annotated with the major block names so that it can be related to earlier figures in this chapter. The lower half is the regular, custom designed, datapath, the order of which is detailed in figure 8-3. The control section is predominately compiled standard cells. The cells are taken from the basic cell library from ARM Ltd. with the addition of the Sutherland elements described in chapter 3. There are two areas of the control that are implemented as PLA structures. These are synthesized using a modified version of the ARM Ltd. PLA generator [Howa89].

Internally, the PLAs are dynamic with a self-timed matched path completion signal which powers down the PLA after the results are latched. The modifications for use in AMULET1 simply made the completion signal available at the PLA interface. The PLA circuit, as used in the ARM6, is a good example of a high volume self-timed circuit in use today and shows that it is possible to have reliable self-timed circuits.

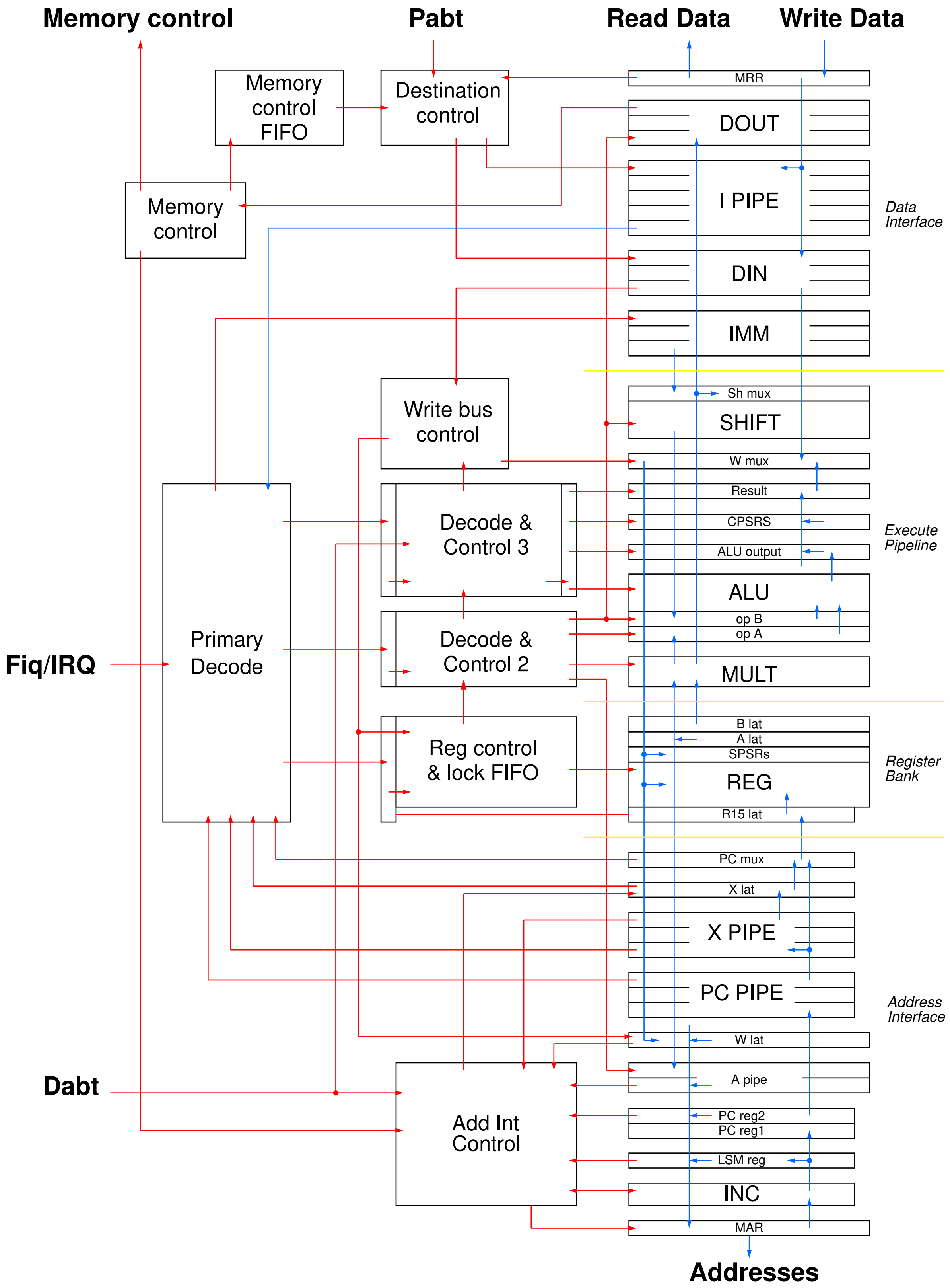


Figure 8-3: Organization of the datapath and control dependencies

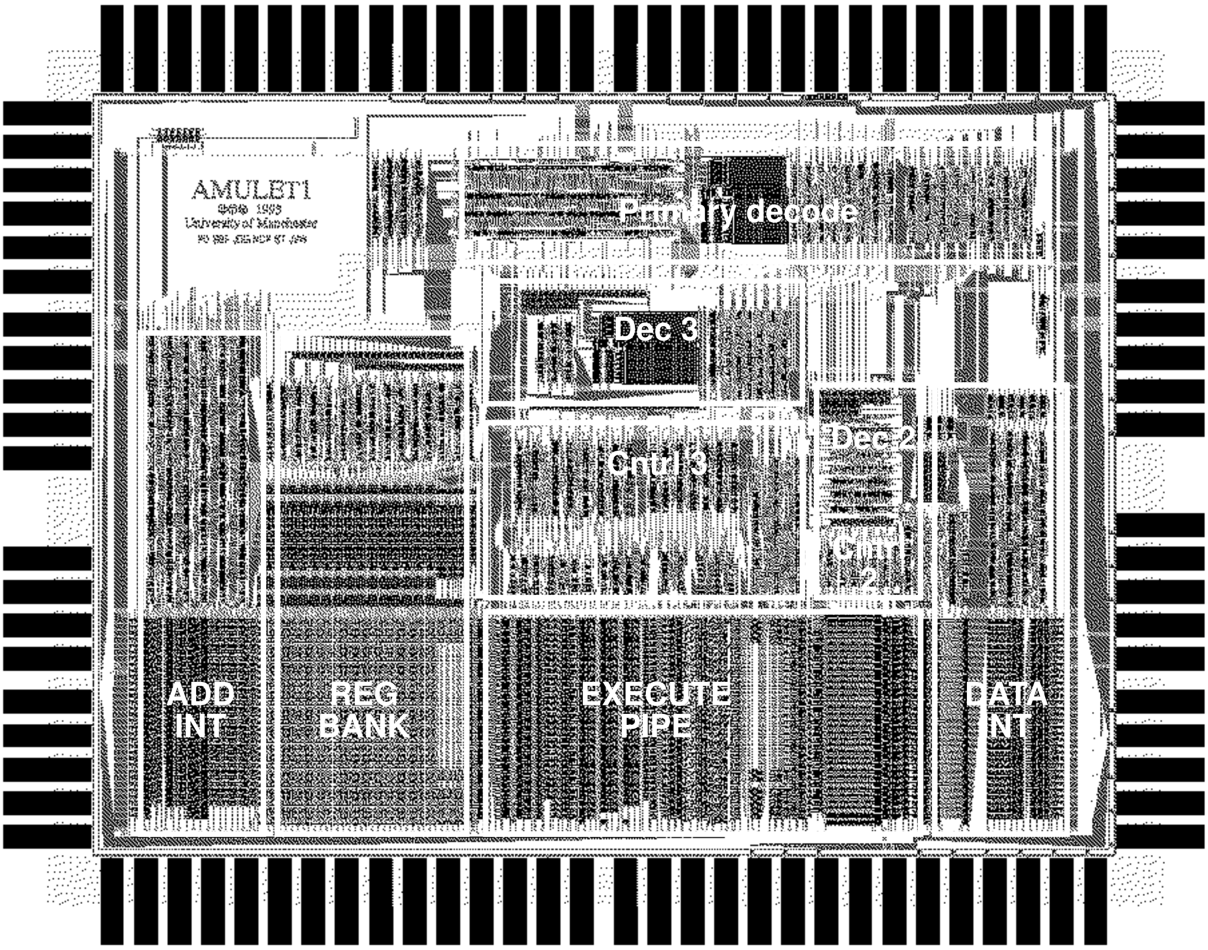


Figure 8-4: AMULET1 1.2 micron physical layout

8.4 Test devices

Although the design of AMULET1 was targeted at the VLSI Technologies CMN12 (1.2 micron) process, the first test devices were fabricated on a 0.7 micron CMOS process by GEC Plessey Semiconductors. The silicon layout was translated onto the new process by a series of semi-automatic geometry transformations applied by the foundry.

AMULET1 was fabricated as part of a multi-project wafer. It shares a single gate array pad ring with other third party test circuits, with only one of the test cores connected to the pad ring in any particular die (the last metal layer determines which core is connected to the pad ring). Figure 8-5 shows a plot (from CAD tools) of the organization of the die with the AMULET1 core clearly visible in the lower left corner.

Figure 8-6 shows a photograph of the fabricated devices mounted in a PGA package. The organization of the die is as in shown the previous figure (figure 8-5) with the AMULET1 core in the lower left corner. All 256 pads are bonded out to the package but the AMULET1 core is connected to only 100 pads.

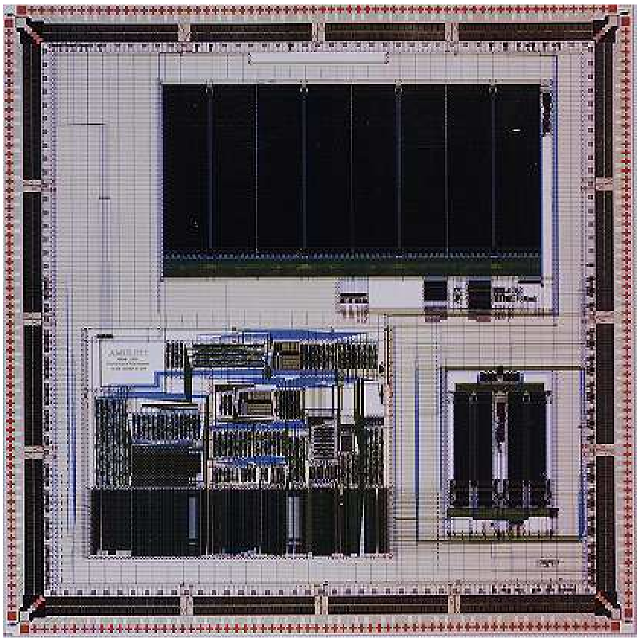


Figure 8-5: AMULET1: 0.7 multi-project die organization

Figure 8-6: AMULET1: 0.7 micron multi-project die

Chapter 9 : Evaluation and further work

Twenty four (0.7 micron) test devices were delivered from the foundry; ten were delivered untested and fourteen had been successfully wafer probe tested (using the static test vectors that were generated from ASIM - see section 8.1.1).

A printed circuit board test card was constructed to evaluate the devices further. The board consists of an AMULET1 device, 128K of RAM, 128K ROM, a UART (for a serial interface) and associated control logic. The test card was designed to be compatible with the standard ARM6 PIE card [ARM92] so that the ARM6 debug monitor[1] (contained in a ROM) could be used to analyse the AMULET1 design.

The debug monitor ran successfully on the AMULET1 test card and enabled the card to communicate with a host machine to download and run programs. All the ARM validation programs which could run on the test card completed successfully[2]. Of the ten untested devices seven were found to be functional, and of the fourteen probe tested, twelve were found to be functional after packaging

Preliminary performance measurements using the Dhrystone benchmark have shown AMULET1 operating at 28K Dhrystones; further refinements to the test card could well improve on this. With no on-chip cache the performance of AMULET1 is likely to be limited by the speed of the test card memory system.

The power consumption of the AMULET1 core cannot be measured separately on the first test devices because the core and pads share the same power supplies (this is a consequence of the multi-project die). Further devices are being fabricated (on a CMN12 compatible process at a different foundry) with the core power supplies separated to enable accurate power measurements to be taken.

9.1 Design characteristics

As a basis for comparison between AMULET1 and ARM6, table 9-1 shows the characteristics of both chips on the CMN12 process, using performance and power figures obtained from simulations under *slow-slow* conditions.

1. ARM60-PIE DEMON V1.0, ARM Ltd., 1991.
2. Some of validation programs are written in such a way that they must be loaded at address location 0 in memory and therefore cannot run under the debug monitor because it also uses the same memory area.

Table 9-1: Characteristics of the AMULET1 compared with ARM6

	AMULET1	ARM6
Process	1.2 μm DLM CMOS[a]	1.2 μm DLM CMOS
Cell core area	5.5mm x 4.1mm	4.1mm x 2.7mm
No. of transistors	58,374	33,494
Performance	9 K Dhrystones[b]	14K Dhrystones @ 10MHz
Dissipation	83mW[c]	75mW @ 10MHz
Design Effort (approx.)	5 man years	5 man years

a. Double Layer Metal CMOS
b. Simulated performance under slow-slow conditions
c. Estimated power consumption from simulation

This table shows that the silicon implementation of AMULET1 is approximately twice the cell area of an ARM6 on an equivalent process. This is also reflected in the 75% more transistors used in AMULET1 compared to ARM6.

The performance figures for AMULET1 shown in table 9-1 are based upon worst-case[1] simulation of the device running Dhrystone code compiled using a standard ARM compiler. The ARM6 figures are the worst-case specification from the data sheet[2] [ARM91b]. The difference between the estimated performance (shown in table 9-1) and that achieved by the test devices can be accounted for by observing that the supplied devices represent *typical* silicon[3] which is usually twice the performance of *slow-slow*, and the transition from a 1.2 micron process to a 0.7 micron process results in approximately 30-50% speed improvement (this figure is hard to quantify exactly).

The power dissipation figures for AMULET1 are taken from a tool which monitors all extracted layout nodes in a simulation and calculates the energy dissipated whenever there is a transition at a node [Davi94]. The power is calculated by summing the energy dissipation at all nodes and averaging this value over time. The results from this power analysis tool have yet to be verified but this is the best estimate for power consumption at present.

The design effort required to design and implement the asynchronous organisation of AMULET1 was comparable to the cost of producing its synchronous counterpart.

9.1.1 Area overhead

The organization of AMULET1 employs a relatively deep pipeline, which accounts for much of the increase in transistor count and die size relative to the ARM6. The ARM6 also

1. Worst-case simulation is when both n- and p-transistors are characterised as slow and the process is known as *Slow-Slow*.
2. Production ARM6 devices have been qualified at significantly higher clock rates.
3. GEC Plessey Semiconductors have measured the silicon and classified it as *typical*.

has more compact silicon layout (approximately 14% of the AMULET1 die area is under utilised).

The area overhead of the asynchronous control of AMULET1 can be estimated by considering a synchronous implementation of a similar pipeline organization. The datapath part of the design would remain broadly similar but the control for the latches etc. would be driven from the global clock rather than individual control circuits communicating between themselves. The control area overhead can therefore be derived by calculating the area consumed by these individual control circuits and their event communication paths. A first order approximation of this cost can be given by assuming all event control modules form the overhead. Table 9-2 shows a breakdown of the event control module use in AMULET1 and the percentage of the utilised core area they use.

Table 9-2: AMULET1 event control module area overhead

Module	Number	% of Total Core Area
XOR	139	0.94
CGate	98	0.86
TOGGLE	67	2.9
SELECT	56	2.15
CALL	13	0.52
ARBITER	5	0.28
DWAIT	7	0.22
Total module area		7.87
Wiring cost		7.87
Total area overhead		15.74

As well as the area of the modules themselves there is also an overhead associated with their wiring interconnect. By examining the existing standard cell areas in AMULET1 it can be noted that each row of standard cells has, on average, a similar area of wiring interconnected associated with it. Therefore the estimated wiring cost given in table 9-2 is the same as the module area. The table shows the total estimated cost of the asynchronous control to be approximately 16% of the total core area of AMULET1.

9.1.2 Pipeline organization

Normally it would be expected that a deeper pipeline would result in an increased throughput, however in AMULET1 this is not the case because of several factors:

- The primary decode takes too long to process instructions, so the pipeline is never busy.
- Register dependencies between consecutive instructions cause the pipeline to stall.

- The depth of the pipeline increases the cost of branches.

Time and resource pressure in this first asynchronous design did not allow for the primary decode to be modified once it was realised that it represented a bottleneck, but this can be addressed in the future. The problem of register dependencies is a function of the way the compiler allocates registers and schedules instructions. The standard ARM compilers take no account of inter-instruction dependencies because the performance of the ARM6 is not affected by them, however the performance of AMULET1 can be adversely affected by inter-instruction dependencies. To achieve optimum performance the compiler must therefore schedule instructions to avoid dependencies between consecutive instructions where possible. Preliminary investigations have shown that it is possible to improve the performance of AMULET1 significantly by including such compiler optimizations.

The cost of inter-instruction dependencies remaining after these compiler optimizations have been applied can be reduced by modifications to the processor organization to include the last result re-use and register through-passing techniques described in chapter 5.

Pipeline depth

Retrospective analysis of the AMULET1 design has revealed that the depth of pipelining is too great. This is partly due to FIFO buffers being conceptually easy to use within the Micropipeline design style, and as a result too many were added. There are many stages that contribute little (or nothing) towards performance but still cost silicon area, transistors and power dissipation (and some stages actually decrease performance!).

The depth of the pipeline also has an adverse effect on the branch latency[1]. As branches represent approximately 20% of ARM instructions [Furb89] (i.e 1 in 5 instructions) the increased latency significantly affects performance.

9.2 Further work

Further work (sponsored by the ESPRIT OMI-DE project) is already under way in several areas. The main objectives of the work are to revise the asynchronous core to reflect the experience gained during the first implementation and to add an on-chip cache and MMU. The aim of the core modifications are to improve its performance although there is much work to do before an exact figure for the extent of this improvement can be given. Some of the proposed modifications are discussed here and are split into four sections: base technology, processor organisation, development tools and test.

9.2.1 Base technology

In chapter 3 we saw that there were trade-offs in different latch styles in terms of area and speed. Preliminary investigations have show that is possible to improve performance and reduce power consumption by adopting a Svensson [Sven89] style latch, as shown in figure 9-1. This latch requires only a single enable signal (c.f. transmission gate latches which require the true and complement of the enable signal) so simplifying the control circuitry required. The enable transistors can be much reduced in size in comparison to the standard

1. The branch latency is defined to be the time from a branch beginning execution until the time an instruction from the branch target begins execution.

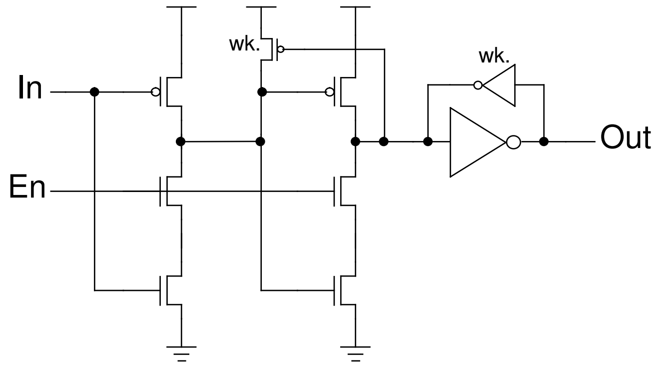


Figure 9-1: Svensson style latch

latch and so the capacitive load on the enable line is reduced. Both of these factors reduce power consumption and improve performance. This work is being undertaken in conjunction with the OMI-HORN project [Day93].

Investigations are also under way to determine the cost of two-phase signalling in relation to four-phase. To achieve this, the primary decode is being re-implemented in four-phase logic so that a realistic comparison can be made.

9.2.2 Processor organization

There are several areas of the processor organization which will be revised in future versions of the AMULET processor. The technique of last result re-use and register through passing described in chapter 5 will be included in new designs to reduce the cost of inter-instruction dependencies.

The primary decode in AMULET1 is one of the major performance bottlenecks in the current design. Time pressure in the original design did not allow this to be modified but future versions will incorporate improvements in this area.

Many of the pipeline stages in AMULET1 contribute nothing to the overall performance of the design, therefore future version of the asynchronous processor will have a much reduced pipeline structure with some pipeline stages merging. (Current investigations are considering the viability of merging the shift/multiply stage with the ALU stage. Logic would be provided to bypass the shifter when it is not required or a shift of zero is specified).

Support for a co-processor interface will be added in future versions to enable an on-chip cache controller and memory management unit to act as a co-processor. There are no plans to add an external co-processor interface at present.

9.2.3 Tools

Work has begun to investigate the compiler optimization strategies which would be appropriate for the asynchronous organization of AMULET1. This involves taking an existing compiler (GCC [Stall92]) and modifying the code generating “back-end” to perform the required optimization. An example of the type of optimization being considered is to reduce the inter-instruction register dependencies of adjacent instructions by code re-ordering. If the number of register dependencies is reduced then the register bank will stall less frequently on register locks so the performance should increase accordingly.

There is a need to develop formal tools to help analyse potential deadlock situations and prevent them occurring. To this end, a high level model of the entire processor is being constructed in Occam as part of other research [Theo93]. The Occam model is a more appropriate starting point for any possible formal reasoning than the detailed transistor schematics.

To verify the integrity of the layout produced, it would be convenient to have tools which could verify statically that the bundle constraints have been satisfied. This could operate in a similar manner to a standard synchronous timing verifier [Comp91d]. Given the completion circuitry and the data circuitry it should be possible to check statically that the bundle constraints are satisfied provided there is no internal feedback. As Micropipelines operate in fundamental mode (section 1.2.2), all feedback must flow through a latch stage (and hence a Micropipeline interface). This means it is possible to break all feedback loops if the correct level of abstraction is chosen.

9.2.4 Test

One of the least explored areas that needs addressing in future work is that of test and verification. The design of AMULET1 incorporated a very ad-hoc approach to test. Static vectors were generated from the high-level model by placing large delays in the event control wires passing through the test interface. By this means it was possible to generate static vectors for the complete chip, so the resulting device could be tested by a conventional tester.

Further research has begun in conjunction with the University of Hanover to investigate the testability of asynchronous circuits (University of Hanover is a partner in OMI-DE).

Chapter 10 : Conclusions

The design of AMULET1 has achieved the objectives set out for this work by demonstrating the feasibility of implementing a complex commercial RISC architecture using asynchronous design techniques. The resulting design addresses many of the difficult issues associated with modern RISC processors such as support for exact exceptions, backwards instruction set compatibility and pipelined operation.

The AMULET1 design exhibits several innovative features:

- The incorporation of a simple ripple-carry asynchronous ALU adder with a data dependent propagation time which performs better, on average, than more elaborate carry select adders.
- A novel arbiter-free register coherency mechanism which allows register read and writes with arbitrary timing and also allows internal ALU cycles to overtake slower external memory accesses (assuming there are no register dependencies between the two).
- An instruction prefetch unit which has a non-deterministic (but bounded) prefetch depth beyond a branch.

The design also includes a pipeline structure which allows asynchronous concurrent operation of internal functional units.

10.1 Micropipelines

Micropipelines appear to offer a good engineering framework for the design of an asynchronous microprocessor and the design details have much in common with synchronous design. The re-use of synchronous library elements is a major advantage of the Micropipeline approach over other asynchronous design styles, especially for data processing operations where many bits are processed in parallel and area and efficiency are important. In AMULET1 it was possible to use the existing silicon layout of the ARM6 register cells and barrel shifter simply by adding a self-timing path to each. This yielded a very efficient, compact datapath layout with a comparable cost to the synchronous implementation. The overhead of using the asynchronous Micropipeline control has been estimated to be approximately 16%.

The flexibility of the asynchronous Micropipeline approach allows designs to be optimized for typical operating conditions and allows resources to be concentrated on functions that are used most frequently with rare worst-case functions being allowed more time to evaluate (c.f synchronous design where resources are used to speed-up the rare worst-case functions). The asynchronous design style does, however, prevent some of the architectural enhancements often associated with deeply pipelined clocked processors (e.g. register forwarding) from being used, but alternative solutions to these problems can be adopted (e.g. last result re-use). The Micropipeline design style makes FIFO buffers conceptually easy to use, however care must be taken to ensure that their ease of use does not lead to them being used too liberally (as was the case in AMULET1).

The robustness of the Micropipeline design style has been demonstrated by the fact that a circuit which was originally designed for a 1.2 micron process has been successfully translated onto a 0.7 micron process (with all process translations being performed semi-automatically at the silicon foundry). The effort required to design and implement AMULET1 using the Micropipeline methodology was comparable with that required for the synchronous processor (5 man years).

10.2 AMULET1

The asynchronous design is within a factor 2 of the synchronous ARM6 in the important parameters of performance, silicon area and power consumption, but at present it does not show any significant advantage over its clocked counterpart. However, AMULET1 is a first attempt at a Micropipelined design of this complexity and scale, whereas the ARM6 is a fourth generation synchronous processor and a world leader in its class for small die area and power-efficiency. In addition the ARM instruction set contains many features that were defined as an artifact of the original synchronous implementation. This has caused extra complexity in AMULET1 because of its extended pipeline structure[1] and its asynchronous operation.

The performance of AMULET1 is also degraded by conservative engineering margins; in the first design the emphasis was placed upon functionality rather than outright performance, and there is considerable scope for enhancing the speed and power efficiency of the design in the future.

10.3 An asynchronous future?

Asynchronous design styles are enjoying a resurgence of interest at present due to their immunity from clock skew and their potential for high performance and power efficiency. Many questions remain to be answered before this potential can be fully realised. Up to now, one of these questions has related to the feasibility of designing effective asynchronous circuits at the level of complexity required for commercial applications. AMULET1 answers this question by providing a convincing demonstration that complex asynchronous circuits are, indeed, feasible.

---

1. A synchronous ARM with a similar pipeline structure would also encounter some of the same problems.