Chapter 4 : The asynchronous ARM

To investigate the suitability of the Micropipeline methodology for the design of complex systems, an asynchronous implementation of the ARM processor was developed. The asynchronous ARM (AMULET1) is object code compatible with the 32-bit ARM6 and addresses several difficult areas.

4.0.1 The ARM processor

The ARM processor employs a simple and efficient RISC architecture. It was originally designed at Acorn Computers, Cambridge, England in 1983-84. Its main features are:

- A small and simple design with fewer than 35,000 transistors.
- A RISC load/store architecture, with support for efficient block data transfers.
- All instructions can be conditionally executed.
- Low cost because of its small size.
- Very low power, again because of its simplicity and small size. (One of the marketing metrics used by ARM Ltd. is that it delivers 100 MIPS/Watt).

The motivation behind the original design and a detailed description of the architecture are described elsewhere [Furb89, VLSI90], a brief overview of the processor is presented in appendix B.

4.0.2 Implementation challenges

The asynchronous implementation was required to be code compatible with the 32-bit ARM6 and therefore had to address the following difficult areas:

1. Interrupts - The ARM supports two levels of interrupts known as IRQ (Interrupt ReQuest) and FIQ (Fast Interrupt reQuest). Both use level sensitive input signals; FIQ has higher priority.
2. Exact exceptions - The ARM processor supports virtual memory (VM) systems and is therefore able to handle data aborts, for example, caused by a page fault in a VM system.
3. Block data transfer - The ARM can load or save multiple registers from the current working set with one instruction. This requires support for multi-cycle instructions.
4. Conditional instructions - Every ARM instruction is conditionally executed depending on the state of the processor status flags.

The above list presents some of the more challenging features, each of which on its own is non-trivial; handling all of them together can become very complex. For example, a data abort can occur in the middle of a load multiple operation where half the registers have already been reloaded before the exception is raised. The load multiple instruction must be re-started after the cause of the exception has been removed. This is particularly difficult when the base register (which specified the memory address used in the transfer) has itself already been overwritten (e.g. in a context restore) before the exception was flagged.

4.0.3 Differences from the ARM6

The asynchronous organization of AMULET1 differs from the ARM6 in the depth of pipelining employed. The ARM6 has a three stage pipeline (see appendix B): fetch, decode and execute, whereas AMULET1 employs a much greater depth of pipelining. In particular, the execute phase of AMULET1 is sub-partitioned into a further three pipeline stages (register read, shift/multiply and ALU). The exact detail of the pipeline structure is discussed later.

A small number of architectural features were not implemented or do not conform exactly to the ARM6 specification due to the limited resources and time available. The features not implemented include:

- Co-processor instructions.
- The ARM6 26-bit compatibility mode which allows the ARM6 to behave as if it was a 26-bit ARM2.
- The multiply-with-accumulate instruction *MLA* (although multiply without accumulate is implemented).

All the co-processor instructions take the undefined instruction trap and their functionality is emulated in software by the trap handler (the first synchronous ARM also used this approach).

There are several obscure corners of the instruction set where the asynchronous implementation deviates from the specification because the particular “features” were defined as an artifact of the synchronous implementation and have little or no practical use.

4.0.4 Processor interface

The asynchronous implementation of the ARM employs an asynchronous Micropipeline interface to the environment. Figure 4-1 shows how a system containing an asynchronous ARM is configured. It is assumed that the input/output forms part of the memory system. The memory management unit (MMU) and the memory have not been implemented as VLSI circuits at this stage and are beyond the scope of this thesis. The asynchronous interface of AMULET1 has the following signals:-

- The output bundle which contains a memory address, control information and write data, if a write operation is being performed.
- The input bundle which returns read data requested from memory back to the processor.
- The interrupt requests, memory abort response and processor reset.

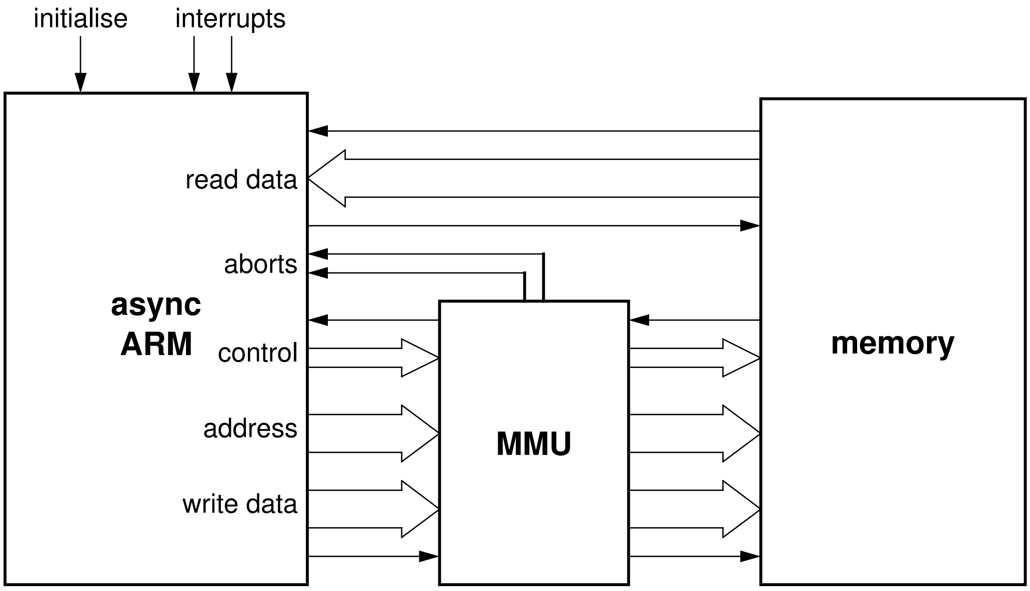


Figure 4-1: The processor interface

The control information in the output bundle consists of a number of bits which include read enable, write enable, a bit to indicate whether an instruction or data is being fetched (opcode flag), a privilege mode bit so that memory protection can be implemented and 2 bits indicating whether sequential address behaviour is likely (and hence whether the memory access can use the faster page mode of DRAM and some cache memories).

The processor design makes no assumption about the memory hierarchy employed. The only constraint that must be enforced is the sequential ordering of memory accesses so that data returns to the processor in the requested order. The time taken for a memory access is not critical for correct operation, if the Micropipeline protocol is obeyed. For example, a system with a cache memory would exhibit a fast response for a cache hit and a much slower response for a miss. However, cache hits must not be allowed to overtake cache misses or the sequential ordering constraint would be violated.

The ARM6 architecture requires support for virtual memory. The method used to handle page faults is that of an exact exception. For every data access, the state of the processor is preserved until it is known that the access will be successful. For each data access, the MMU produces a fault/no fault response that signals back to the processor either to cause instruction processing to resume (if there was no fault), or to enter the exception handling software, (if there was a fault). The response time of the MMU determines how soon the processor can resume executing instructions, so a fast response time is required for optimum processor performance. The response from the MMU is signalled back to the processor in a dual-rail encoded format.

Aborted instruction-prefetch accesses need not signal back to the processor in the same way, as they do not affect the current state of the processor. The non-valid instruction data can be tagged as invalid when read into the processor. The exception handler is then called when the instruction reaches the primary decode.

AMULET1 also supports the two level-sensitive interrupts of the ARM6 so that it can be used with conventional peripheral chips. The level sensitive model of interrupts gives the processor no information regarding the time taken for the peripheral to release the interrupt line, so care must be taken in the exception handling routines to ensure that the same interrupt does not cause multiple exceptions.

The final input to AMULET1 is the initialisation pin. This is used to reset the state of the processor; its release causes the processor to begin issuing sequential instruction addresses to the memory starting from address zero.

4.1 Processor organization

The internal organization of AMULET1 is shown in figure 4-2. This shows the four main areas of the processor and how they are interconnected. A brief description of each of the areas is presented in this section; more detail is provided later in the thesis.

4.1.1 Address interface

The main function of the address interface is to issue instruction prefetch requests to maintain a steady flow of instructions into the processor. The address interface can autonomously issue sequential instruction addresses with the aid of an internal incrementer.

Data transfer operations use the address interface to generate the data transfer address and the multiple data transfer instructions also use the incrementer in the address interface to generate the sequential data addresses required.

The address interface supplies a PC value to the instruction currently being executed to be used as an operand (via the *PC Pipe*). This is needed because the program counter forms the 'general purpose' register *R15* in the ARM architecture. The address interface is described in more detail in chapter 6.

4.1.2 The register bank

The register bank contains 30 general purpose registers, sixteen of which are accessible at any one time. It also provides storage for the 5 Saved Processor Status Registers (SPSRs) as specified in the ARM architecture definition. To access the contents of the register bank, two read ports are connected to the *A* and *B* operand buses; a single write port is provided for modifying the contents. Internally the register bank has mechanisms for supporting multiple outstanding write operations and managing inter-instruction dependencies. The detail of these mechanisms is described in chapter 5.

4.1.3 The execution unit

The execution unit is the computational core of the processor. The data read from the register bank can be multiplied using an autonomous 3-bits-at-a-time carry-save-adder multiplier. If multiplication is not required then this logic is simply bypassed. The ARM architecture specifies that one of the register operands can optionally be shifted for certain classes of instruction. To support this, a barrel shifter is attached to the *B* operand bus and its output fed to the ALU. The ALU performs all other logic functions and contains a simple ripple-carry adder with data dependent completion signalling. The output of the ALU can be transferred onto the write bus to route results back to the register bank. The write bus is

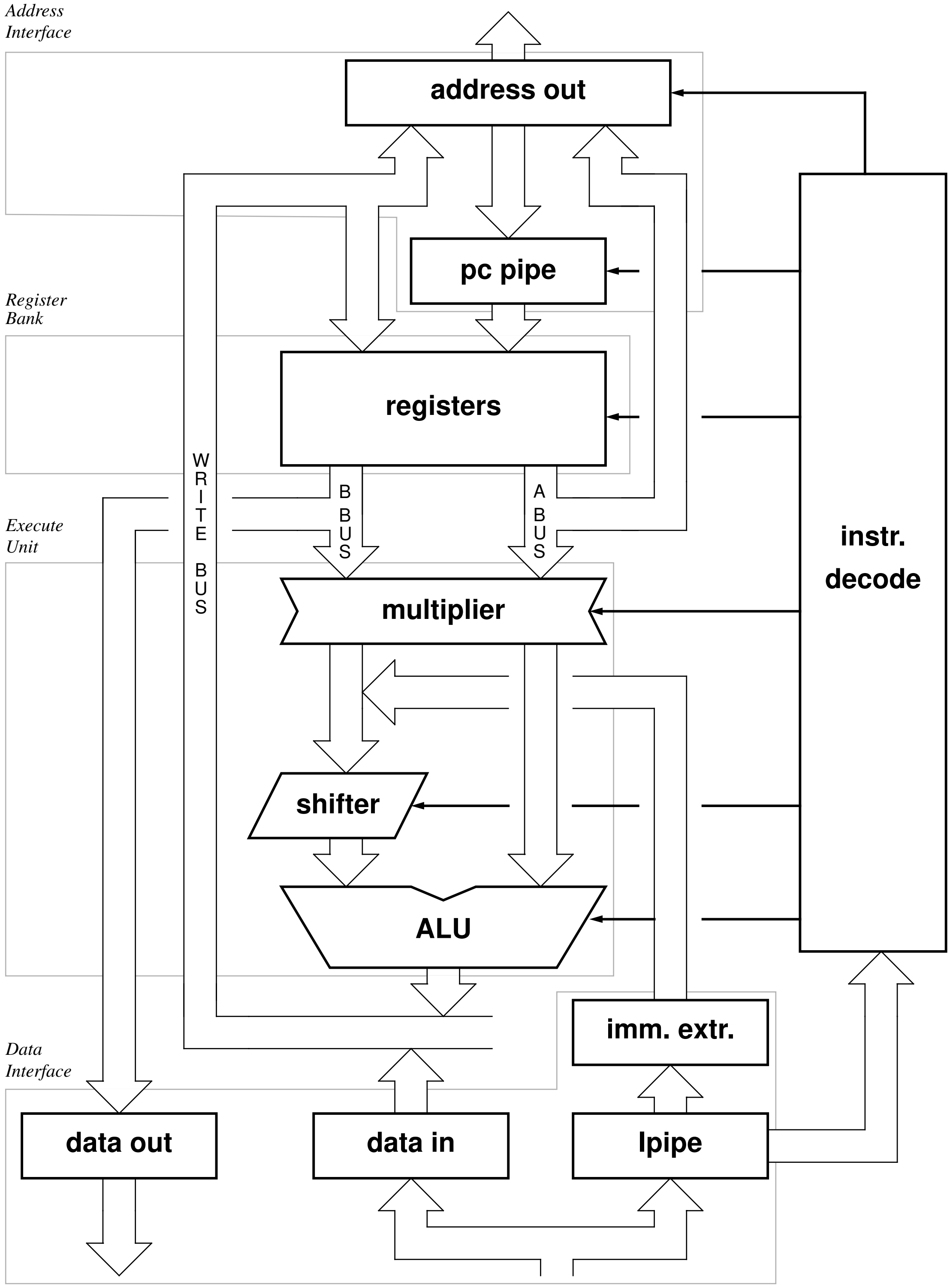


Figure 4-2: Processor organization

shared with the data interface which uses it for incoming data from memory; control of the bus thus requires arbitration (data arrives asynchronously to the ALU result). The execution unit is described in more detail in chapter 7.

4.1.4 The data interface

The data interface manages the flow of data between the processor and the memory subsystem. For write operations it provides a FIFO for storing data waiting to be written to memory (*data out*) and has the capability to replicate the least significant byte into all byte positions across the word (for byte write operations).

Incoming values can be either new instructions for decoding or data destined for the register bank. Instructions are stored in a FIFO prior to execution (*Ipipe*). If the instruction specifies that an immediate value is required then a copy of the instruction is passed to dedicated hardware (*imm. ext.*) which extracts the immediate value from the instruction word.

Data destined for the register bank passes through the *data in* section. This can rotate the incoming data word by byte quantities. Data loads from non-word aligned addresses are performed by loading from the word boundary and then rotating the data word until the addressed byte is in the least significant byte position. *Data in* also contains logic to enable the processor to operate in "little-endian" or "big-endian" modes.

4.2 Pipeline organization and control

Figure 4-3 shows how the processor is divided into pipeline stages; each grey box represents a pipeline latch. The execution phase is partitioned into three stages: register bank, shift/multiply and the ALU. Each data processing stage has a corresponding decode and control stage.

During instruction execution primary decode performs the complete decoding for the register bank and the initial decoding for execution stages 2 and 3. Once the primary decode is complete, the control information is latched along with the corresponding value of the PC. While the operands are being read out of the register bank the secondary decoders for stages 2 and 3 decode the instruction further.

Although the control and data latches are drawn in alignment, the control and datapath only synchronize when they reach their destination stage. Decode 2 only synchronizes with the data when it reaches the shifter/multiplier and decode 3 only synchronizes when it reaches the ALU.

The extra pipeline latch after the ALU is provided to allow the Current Processor Status Register (CPSR) to be accessed; it also decouples the ALU from the result writeback phase.

4.2.1 Dynamic pipeline structure

To reduce their sizes and transistor counts, the register bank, multiplier, shifter and ALU all use dynamic structures. Each stage employs dynamic logic with an output latch for storing the result. The output latch of one stage is the input latch of the next (see figure 4-4(a)). This is similar to the canonical Micropipeline with processing shown in figure 3-6, except that the dynamic logic imposes an additional constraint: the output latch must be empty before

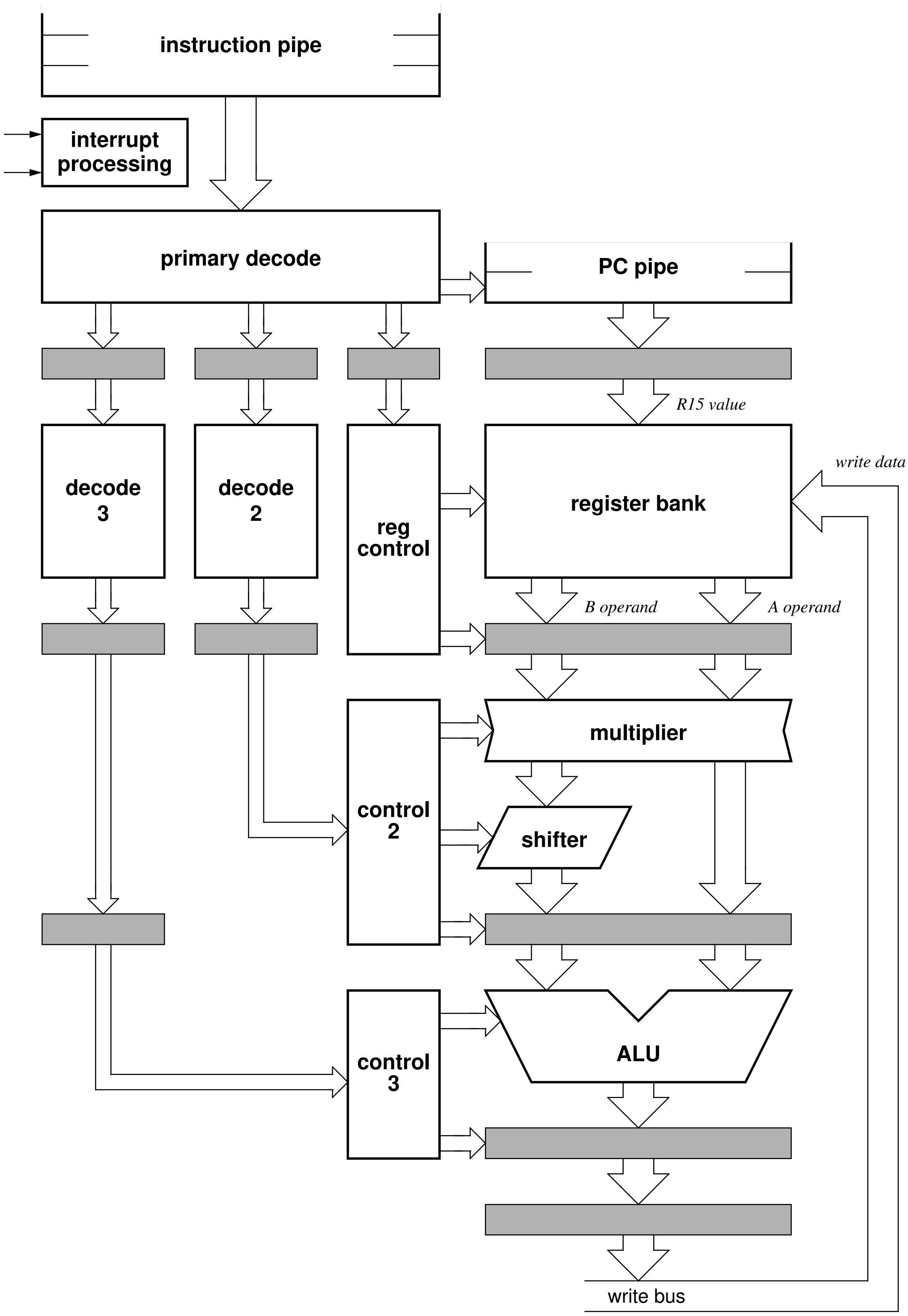


Figure 4-3: Pipeline organization and control

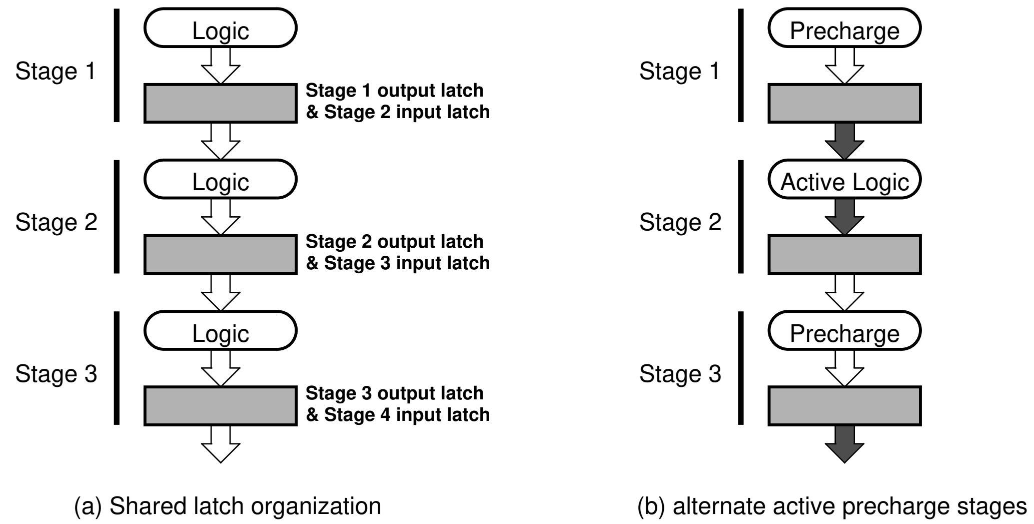


Figure 4-4: Dynamic pipeline structure

the evaluation of the logic begins to ensure the result can be latched before leakage renders it invalid.

When any stage is active it needs both to be able to write into the output latch and also to ensure that the input does not change until the computation is complete, therefore both latches are involved in the activation of the stage. At any instant this means that only alternate stages may be active. (Note, if no results are removed from the bottom stage, the pipeline will backlog with each stage storing its result value in its output latch; in this state the pipeline is fully occupied)

Having only half the pipeline stages active at any one time seems to incur a performance loss; however, the dynamic stages require a precharge phase, so while one stage is active the adjacent stages can precharge (figure 4-4(b)). If an extra latch were provided to decouple the two stages, the pipeline could be fully active because the input and output latches need no longer be shared. However, data flow from one stage to the next would still have to wait for the next stage to precharge. This, along with the extra pipeline latch, would increase the latency through the pipeline.

The dynamic shared latch pipeline with only alternate stages active can be used to advantage in the ALU and shifter. The shifter needs the current carry flag (for rotate with carry) which can be changed by the ALU, but because the shifter and ALU are adjacent stages only one of them can be active at any time. While the ALU is executing and possibly calculating the new flag values, the shifter is in precharge. When the ALU is complete and the flags are valid, the ALU input latch is released and the ALU enters precharge. This allows the shifter to begin execution. The flags are now valid and cannot be changed by the ALU, so they can be used directly by the shifter without the need for complex synchronization between the ALU and the shifter.

4.3 Instruction mapping

The mapping of the ARM instruction set onto the asynchronous organization can be described by examining the datapath activity of each class of instructions.

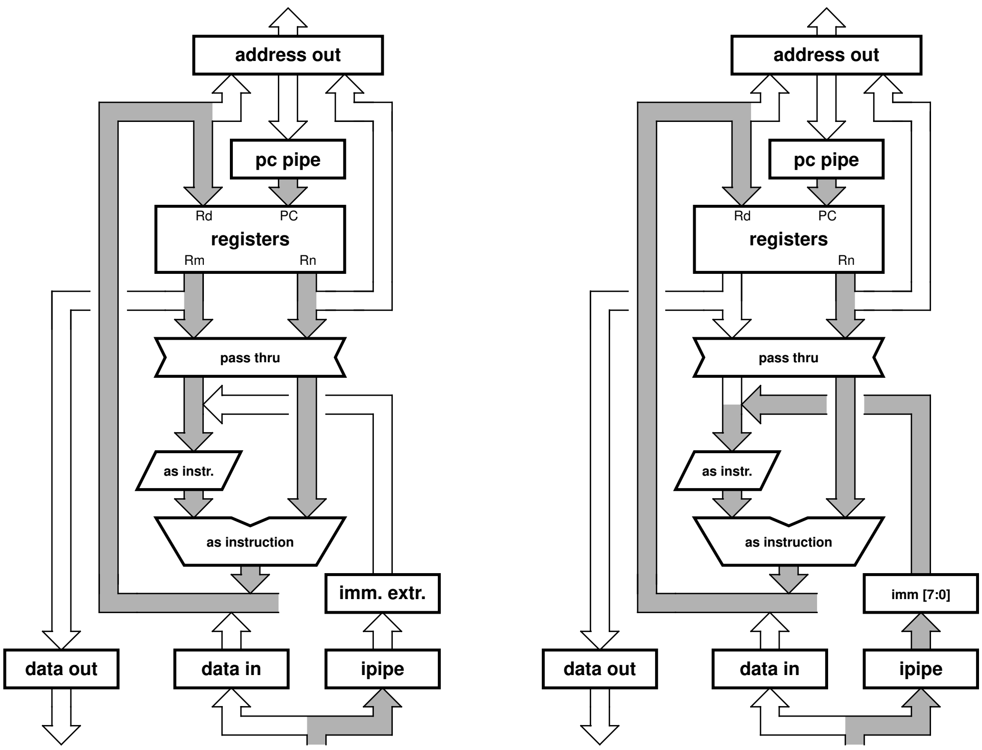
4.3.1 Data operation

The simplest class of ARM instruction is the data processing operations. These can take two operands, perform one of sixteen operations and then write the result to a register.

Figure 4-5(a) shows the datapath activity for a data operation that uses two register operands. Here the two operands are read from the register bank onto the *A* and *B* operand buses (see figure 4-2 for bus naming convention). The multiplier is configured so that the operands bypass the multiplier logic without activating it. The ARM architecture specifies that one of the operands can be shifted by an arbitrary amount so the *B* bus is passed through a barrel shifter to perform this operation (the type of shift performed is specified in the instruction). The two operands then enter the ALU where they are combined according to the data operation instruction being executed. The ALU then takes control of the write bus and sends the result back to the register bank (unless the destination is the program counter[1] - *R15*).

If the destination is *R15*, the result is steered to the address interface instead of the register bank and the operation is similar to that of a branch (described in the next section).

If one of the operands is *R15* then the bottom entry from the *PC pipe* is multiplexed onto the required operand buses. The correct value of the PC is supplied before instruction execution begins, therefore no lock mechanism is needed to prevent register hazards (see section



(a) Data Op with 2 registers (b) Data Op with 1 register & immediate data

Figure 4-5: Data operation datapath activity

1. The program counter can be accessed as a general purpose register (R15) in the ARM architecture.

5.1.2). Any change of instruction flow (and hence PC availability) is managed at a higher level of control.

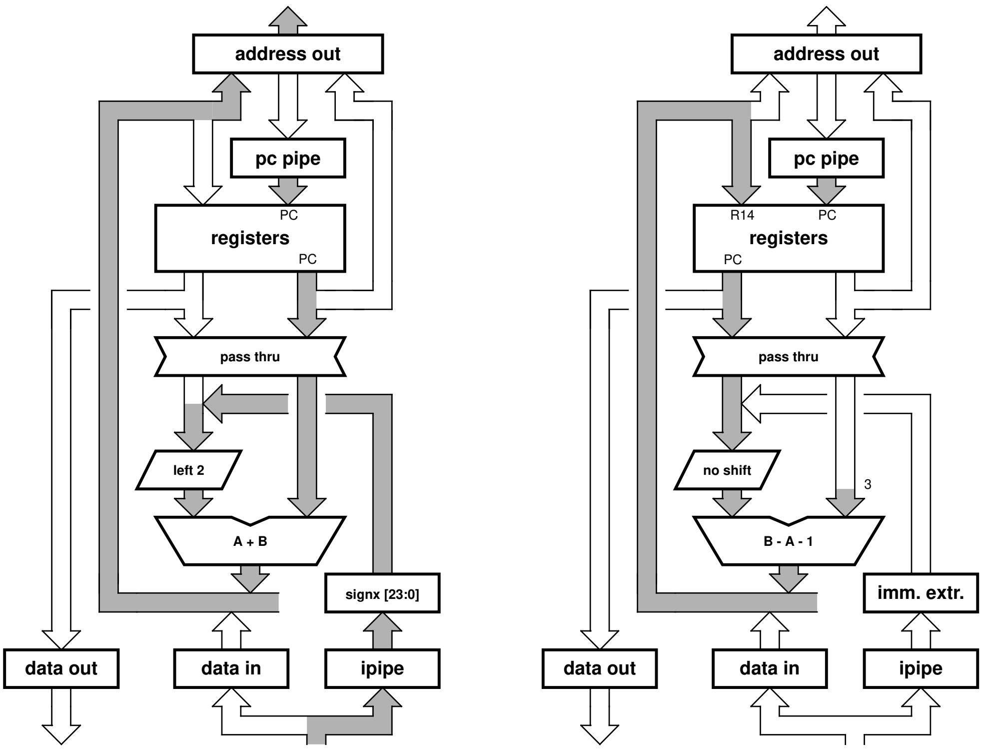
When only one register is required, the other operand defaults to the PC because this will not stall waiting for data to return to the register bank. The PC value is then discarded. It would have been more power efficient to read just one register, but this would have required extra bypass logic to implement whereas the solution adopted requires no additional circuitry.

When one of the operands is an immediate value, as shown in figure 4-5(b), then only one register is read from the register bank. The other operand is an 8-bit unsigned value extracted from the instruction word and then multiplexed onto the input of the shifter. Apart from the source of the operands, the overall operation is as described previously.

4.3.2 Branch operation

In the ARM architecture, the branch (*B*) is a PC relative jump. The instruction word contains a 24-bit immediate offset which is word aligned, sign extended and added to the current PC value to achieve the branch target address. The ARM also specifies a subroutine call instruction, branch and link (*BL*), which is similar to the branch but the address of the next instruction is saved in register *R14* to facilitate the subroutine return.

Figure 4-6(a) shows the branch operation and the first cycle of a branch and link. Here the PC value is read from the register bank onto the A operand bus. The offset is extracted from the instruction word and sign extended to 32-bits. This value is then fed into the shifter



(a) B & BL- change PC

(b) BL- save return address

Figure 4-6: Branch & branch and link datapath activity

where a left shift of 2 is applied to word-align the value before it is added to the current PC value in the ALU. The ALU takes control of the write bus and passes the target address to the address interface. Here the target address enters into the prefetch loop and replaces the original prefetch address (see chapter 6 for further details). Prefetching (not shown on the diagrams) then resumes from the branch target. As a number of instructions may already have been prefetched, these must be thrown away before instructions from the new stream arrive. This is achieved by the instruction *colour* mechanism described in section 4.4.3.

A branch operation is concluded after the above cycle; branch with link however requires an extra cycle to save the return address (i.e. the address of the next instruction). The PC value delivered when *R15* is accessed is PC+8 (see section 4.4.1). The address of the next instruction is PC+4, so the value of *R15* read from the register bank is decremented by 4 before it is written into the link register (*R14*), figure 4-6(b) shows the detailed operation of this. The *R15* value is read from the register bank on the *B* bus and passed through the multiplier and shifter unchanged. Four is subtracted by forcing the bottom two bits of the *A* operand bus High (three in binary) and then performing a subtract with borrow, hence achieving B-A-1 or (PC+8)-3-1= PC+4. The PC+4 value is written back to the register bank via the write bus in a similar fashion to data processing operations.

4.3.3 Multiply operation

The multiplier in AMULET1 uses a shift-and-add technique with carry-save adders (see section 7.1). The multiplier accepts two input operands from the register bank and internally performs the multiplication of the two. When the operation is complete the partial sum and partial carry are placed on the two output buses. These are then added together in the ALU to form the result. This result is written back to the destination register. The datapath activity is shown in figure 4-7.

4.3.4 Load data operation

The data transfer instructions in the ARM allow the programmer to specify a variety of auto-indexing options including pre-/post-index and increment/decrement of an offset from the base address. The operation can be split into two phases; the address calculation and the data transfer. The address calculation will be described first.

Figure 4-8(a) shows a post-index load with a register offset and writeback (in fact, post-index is defined always to write back). Here, the address from which the data is loaded is the value contained in the base register before it is modified. To perform the load, the base register (*Rn*) is sent via a special route directly to the address interface where it interrupts the instruction prefetching to issue the data load address, at the same time the base register is forwarded to the ALU, ready for the base update calculation. The offset register read onto the *B* bus can optionally be shifted before being added to or subtracted from the base in the ALU. The load cycle completes with the modified base value being written back to the register bank.

Figure 4-8(b) shows a pre-index load with register offset and writeback specified. The pre-index designation indicates that the base is modified before being used as the data load address, so the writeback value is the same as the load address. The operation begins with the reading of base and offset from the register bank which are forwarded to the ALU after

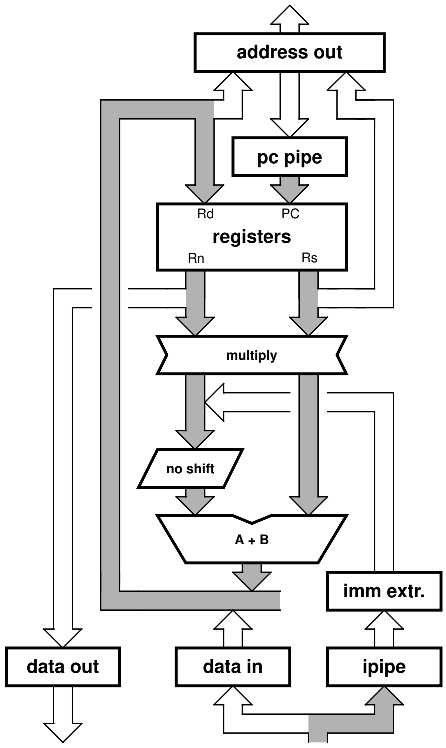


Figure 4-7: Multiply datapath activity

optional shifting of the offset. The base modification calculation is performed and the result transferred to the register bank (if writeback is specified) and to the address interface to be used as the load address.

The ARM architecture specifies that an immediate value can be used as an offset. The datapath activity of the immediate value (not shown) is similar to that shown in data operations except that the offset this time is 12 bits rather than 8 and no shift can be applied to the immediate value.

In the ARM, non-word aligned byte loads are implemented by retrieving the data word from the word boundary and then rotating the value obtained so that the addressed byte is in the least significant byte position of the word. In the ARM6, the rotation is performed in the barrel shifter as the data passes through on its way to the register bank. In AMULET1 instructions which do not depend on the loaded data can continue to be processed while the data is being loaded (see section 5.4.1). This means that the barrel shifter could be busy and unavailable to rotate the incoming data, hardware is therefore provided in the data interface to perform the required byte rotation.

The arrival of data is completely asynchronous to the internal operation of the processor, so when *data in* wishes to use the write bus to transfer data to the register bank, it must arbitrate with the ALU for control of the bus. If data is destined for any register, other than *R15*, it is routed to the register bank (as shown in Figure 4-8(c)). If the destination is *R15* then the loaded data is passed straight to the address interface to be used as the new PC value (as shown in Figure 4-8(d)).

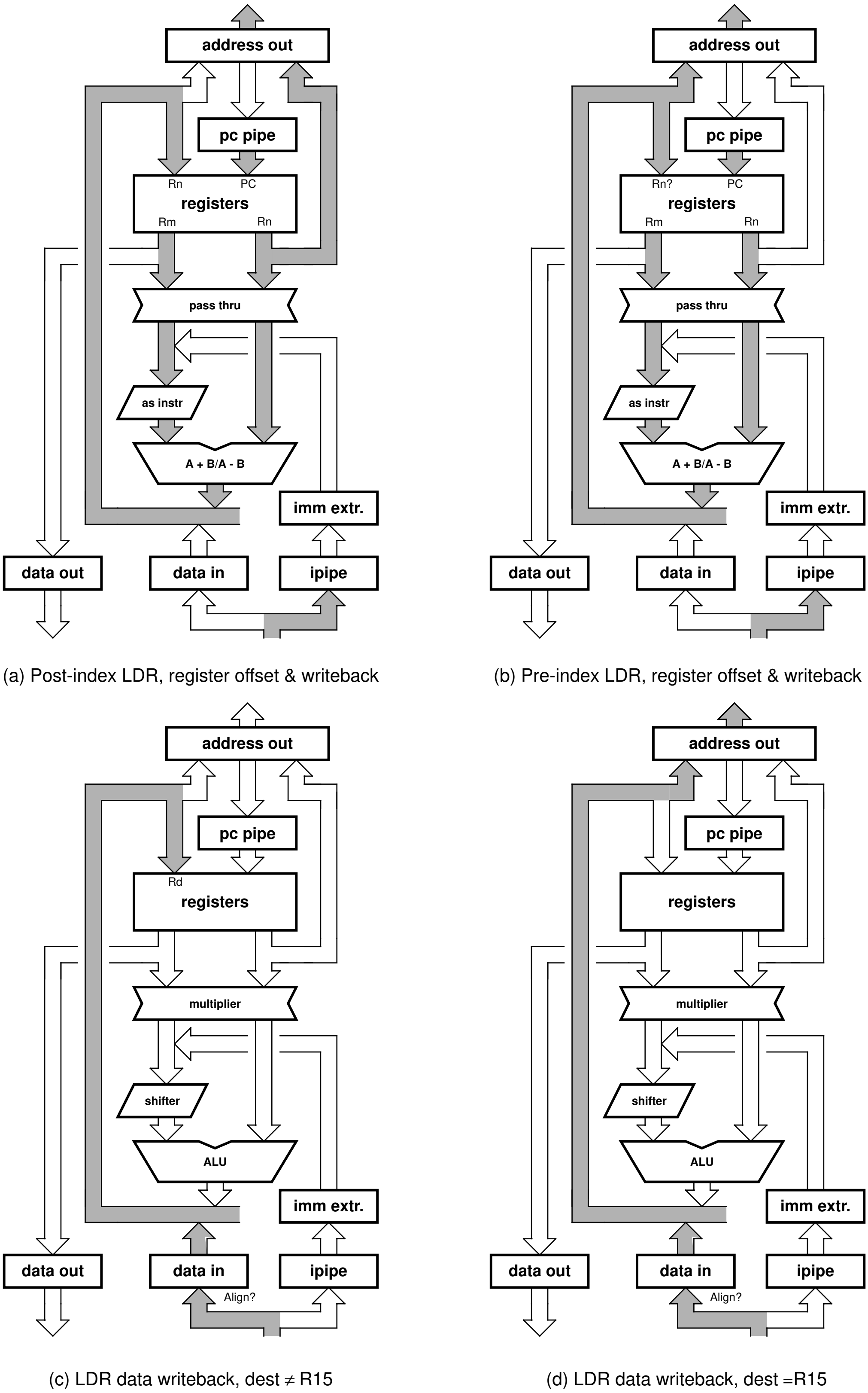


Figure 4-8: Load register datapath activity

4.3.5 Store data operation

The store data operation has the same auto index options as the load instruction class. The single cycle immediate offset store will be described first and then the more complex two cycle register offset case will be analysed.

With an immediate offset, only 1 register (*Rn*) is needed from the register bank for the base address calculation. This allows the other register bank read port to be used to transfer the store data to the data interface (*data out*). The calculation of the address for store with immediate offset is similar to the address calculation described for load operation. Post index store (figure 4-9(a)) uses the direct connection from the register bank to the address interface and pre-index store (figure 4-9(b)) writes the same data to the register bank and the address interface. If store byte is specified, then the least significant byte of the word is replicated in *data out* so it appears at each byte position. The store data then synchronizes with the store address before being despatched to memory for the operation to be completed.

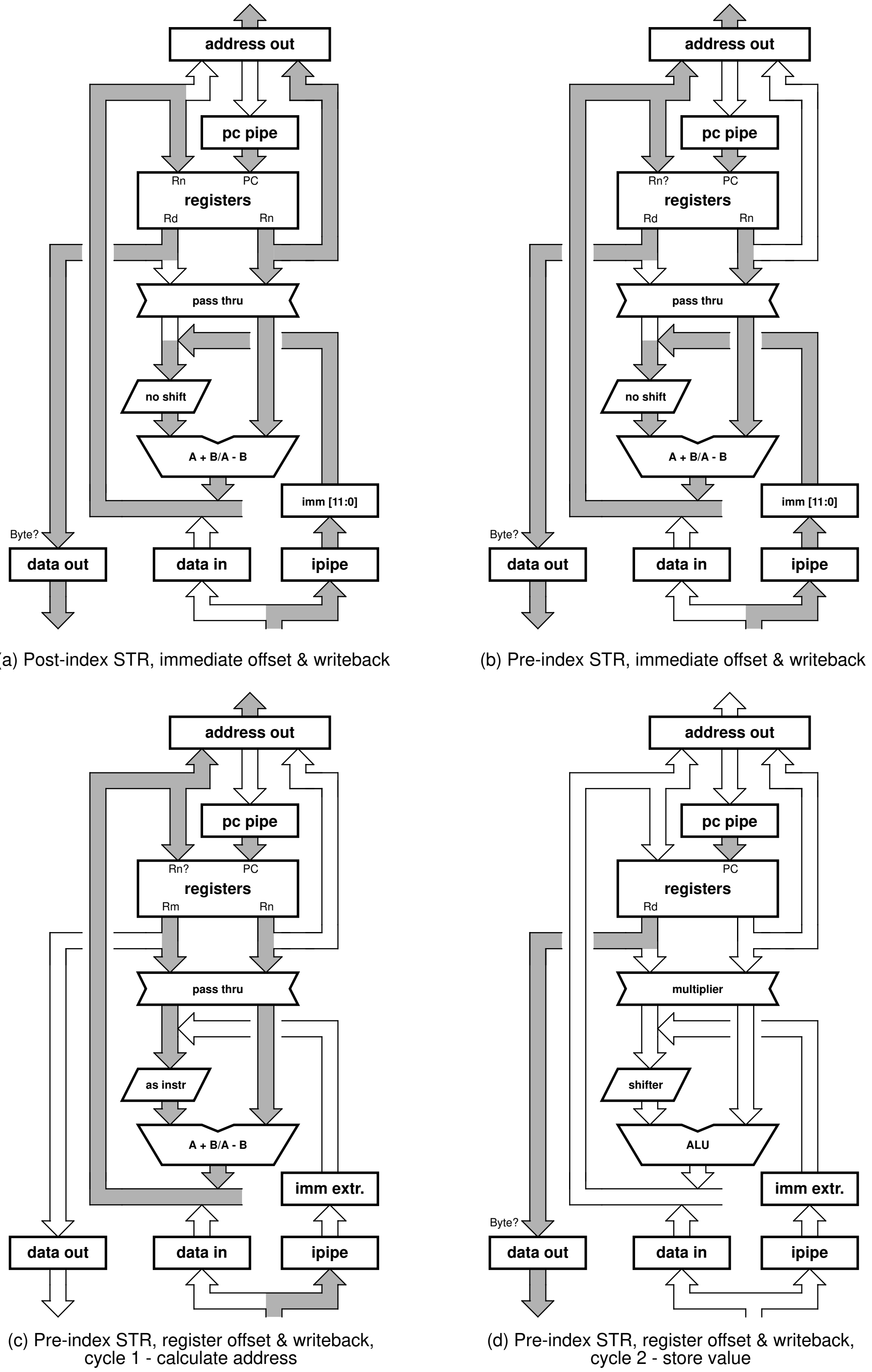
A store data instruction with register offset has a fundamental problem which prevents single cycle operation; three register operands are required but the register bank only has 2 read ports. The three register operands are base, offset and store data. An extra read port could have been added to the register file to make single cycle operation possible but as the store with register offset operation accounts for only 1% of all instructions [Furb89, page 278] the improvement in overall performance would be small. The solution adopted is to perform the base calculation in the first cycle (figure 4-9(c)) and access the data to be stored in the second cycle (figure 4-9(d)).

The cost of the second cycle is hidden to some extent by the depth of pipelining in the execution unit. The store data can be read out of the register bank at the same time as the address calculation is being performed in the ALU. The data route is a simple FIFO (with optional byte replication) so the data propagates quickly and should therefore be available by the time the address is ready to be sent to memory.

4.3.6 Block transfer operation

The block transfer instructions are by far the most complex instructions to implement. The instructions allow any subset of the sixteen available registers to be loaded or stored and the base register to be auto-indexed. The instruction has options allowing different modifications to the base thus supporting various stack paradigms (e.g. empty/full, ascending/descending stacks). The detailed operation of these options is complex and is described elsewhere [ARM91b].

The first cycle of a block transfer instruction calculates the transfer start address. The calculation is performed by reading the base address from the register bank and adding an offset in the ALU (as shown in figure 4-10(a)). The offset is introduced onto the *A* operand bus using a similar mechanism to that described for branch operations. The value of the offset depends whether a load or store operation is being performed and the stacking paradigm selected. For example, an STM will have an offset of 3 for a full ascending stack (add), *Cnt* +3 for a full descending stack (subtract), *Cnt* for an empty descending stack, or 0 for a empty ascending stack (where *Cnt* is the number of registers being transferred).



(a) Post-index STR, immediate offset & writeback

(b) Pre-index STR, immediate offset & writeback

(c) Pre-index STR, register offset & writeback, cycle 1 - calculate address

(d) Pre-index STR, register offset & writeback, cycle 2 - store value

Figure 4-9: Store register datapath activity

Once the address calculation is complete it is forwarded to the address interface where it waits for synchronisation with the primary decode.

The second cycle of the block transfer calculates the value the base register should have after the instruction has completed (i.e. the modified stack pointer). The cycle takes the base out of the register bank and forwards it to the ALU as in the first cycle. The offset introduced this time on the *A* bus is just *Cnt*, the number of registers to be transferred. This is either added or subtracted depending on whether data is being loaded or stored and whether the stack is ascending or descending. The base is then written back to the register bank. For a load instruction, the base value is held in the output latch of the ALU (*old*) in case it is needed for data abort recovery (described below).

The third cycle of a block transfer stalls until it is known that the instruction is definitely going to be executed i.e. it has passed both its conditional and colour tests. This is done because it is very difficult to cancel the data transfers once they have started (data would have to be retrieved from the *data out* pipeline etc.)

To achieve this interlock, the first cycle signals back to the primary decode to indicate whether the instruction will execute. At the end of the second cycle this signal is checked to determine whether the third cycle can begin or must be discarded.

The interlock mechanism could have been placed on the second cycle instead, but this would have caused the execute pipeline to be starved of instructions while the second cycle waited for an acknowledgement from the ALU. By waiting on the third cycle, the second cycle can still proceed and progress down the execute pipe immediately after the first cycle. The cost of the interlock is therefore hidden to some extent because the processor is still executing a part of the instruction. The disadvantage of this approach is that if the instruction does fail its condition test then two cycles have been executed, rather than just one if the more conservative approach had been adopted.

The third phase of a block transfers is the data transfer phase, here the number of cycles is directly related to the number of registers being transferred (one cycle for each register). Figure 4-10(c) shows the data transfer for both load and store multiple instructions. For each register in the data transfer, the primary decode signals to the address interface to generate the memory address of the register being transferred. The next address is generated with the aid of the incrementer in the address interface (the incrementing mechanism is described in more detail in chapter 6). The primary decode generates the address of the register to be transferred.

For store data transfers, the data exits the register bank on the *B* bus and is diverted into the *data out* pipe in the direction of the data interface. The instruction does not activate any further elements on the execution datapath. When the data reaches the data write register it synchronizes with the address generated by the address interface and is dispatched to memory.

For a load instruction, the transfers just involve locking destination registers and signalling to the address interface to generate the load address. The returning data enters the processor in a similar manner to the standard load data instruction.

For both load and store transfers, the signal to the address interface also indicates whether the current transfer is the last. If it is the final transfer then the address is discarded after use as the data address (rather than being incremented ready for the next transfer).

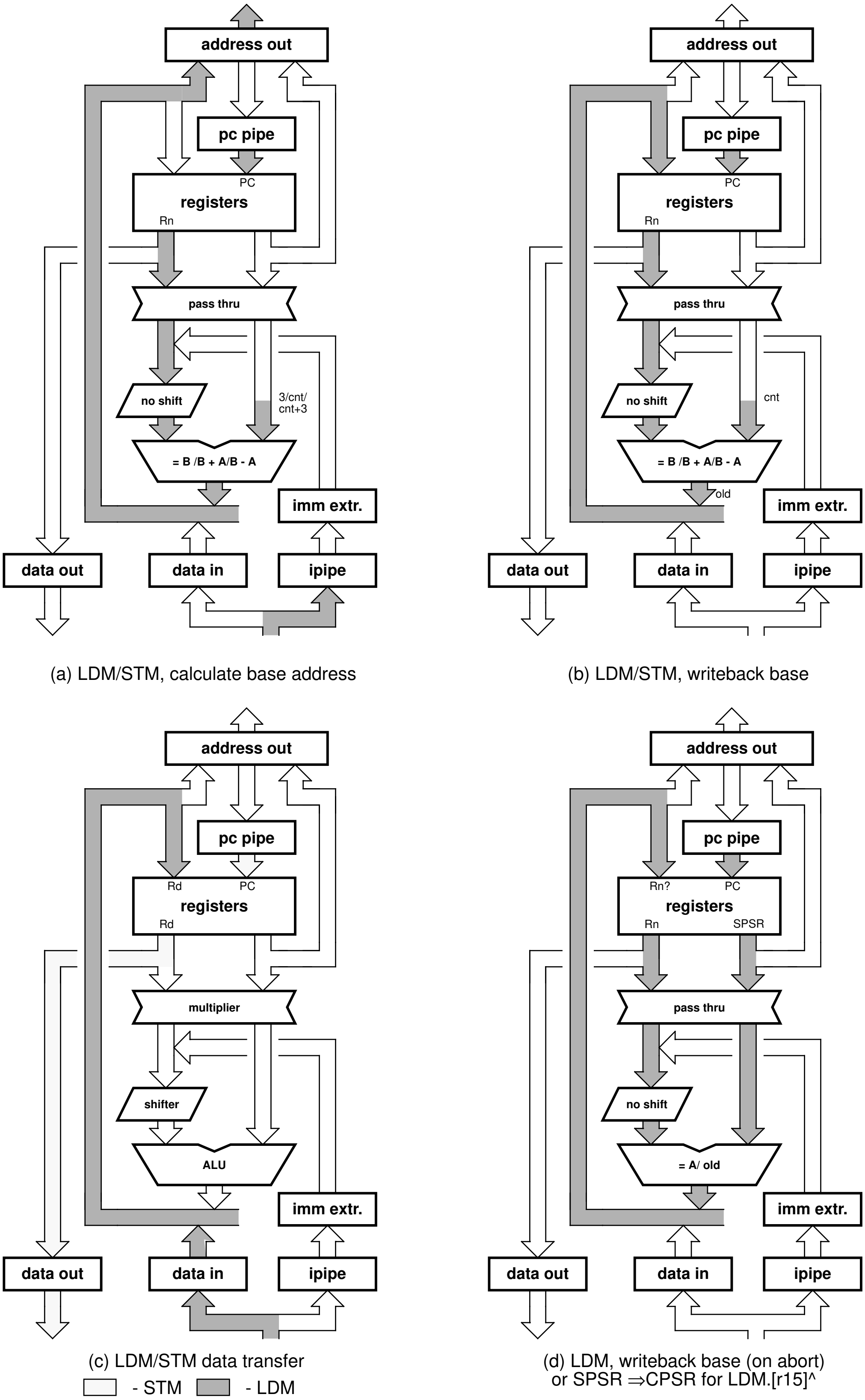


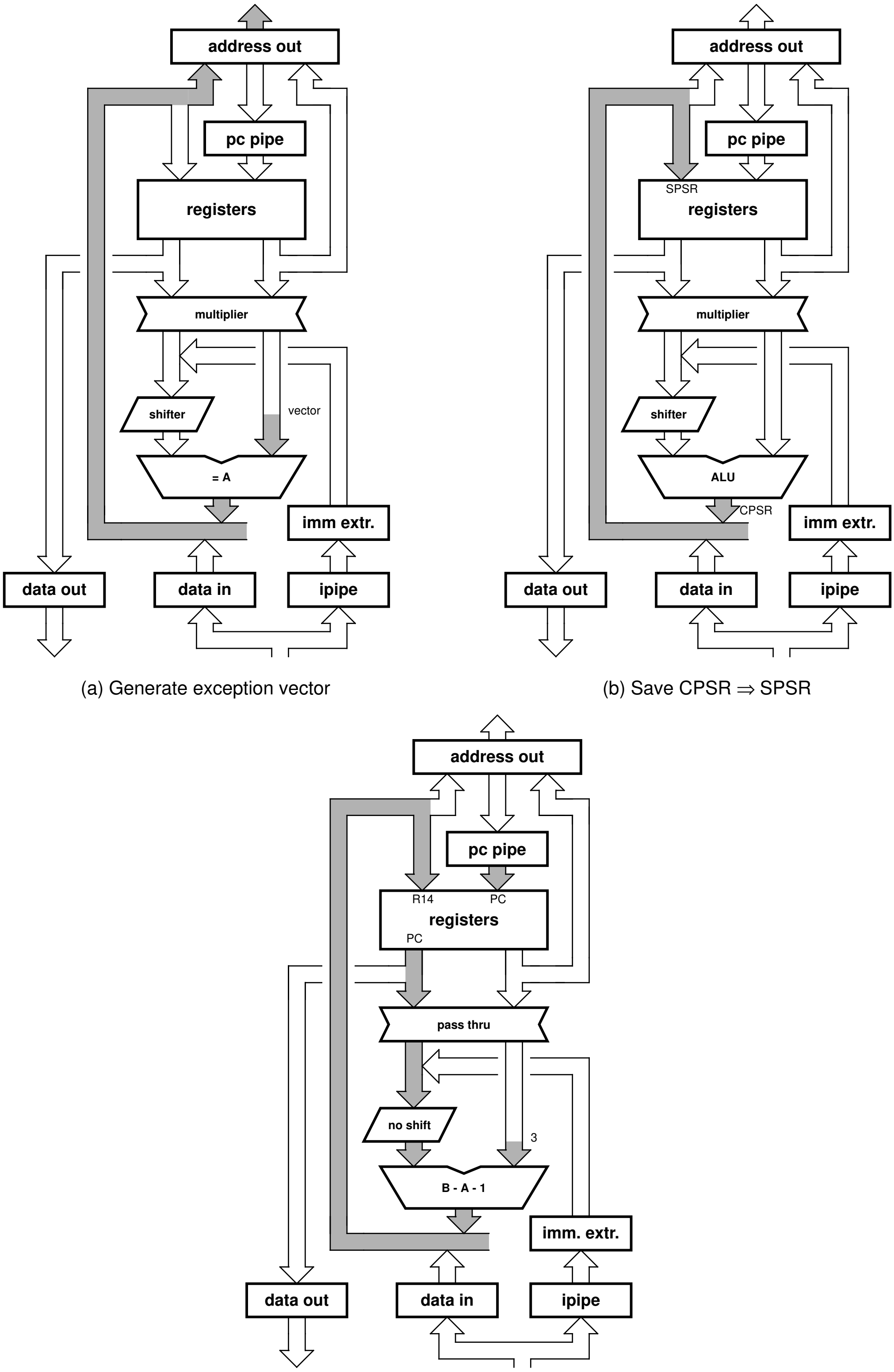
Figure 4-10: Load/Store multiple datapath activity

For load instructions, the signal between primary decode and the address interface also notifies if this transfer is to the program counter (*R15*). This primes the address interface to discard the current prefetch address (after the block transfer is complete) and also forces it to wait for the new program counter, arriving via the *data in* port.

For the store multiple instruction, the transfer is complete after the third phase; however, for load instructions there is another cycle. This final phase (figure 4-10(d)) performs one of two possible functions. The load multiple instruction allows the programmer to specify that *R15* is reloaded with the Current Processor Status Register (CPSR) restored at the same time. The final cycle copies the SPSR to the CPSR in this case. The other possible operation in the final phase is recovery of the base after a data abort. It is possible that the base may have been reloaded before an abort was raised. To restart the instruction, the original base register is needed (although the written back value is sufficient). This is preserved on the output of the ALU in the second cycle (shown as *old*). Therefore, to restore the base, a dummy read from the register bank is sent to the ALU. The ALU ignores its inputs and sends the previously preserved value (*old*) of the base register back to the register bank. It is theoretically possible for this preserved base to arrive back at the register bank before the data for the same register arrives back from memory so the situation can arise that although the base is restored it is still overwritten by the loaded data. To ensure that this cannot happen, the dummy read actually reads the base register, so that if the data destined for the base from memory has not yet returned, it stalls. The read only proceeds when the loaded data is safely in the register, so the base restore can now proceed without fear of being corrupted.

4.3.7 Exception entry operation

Once an exception has been detected by the primary decode, the entry mechanism is the same for all exceptions. The first cycle shown in figure 4-11(a) sends the exception vector address to memory. This is done by performing dummy reads from the register bank and then introducing the exception vector onto the *A* operand bus. This vector is passed straight through the ALU and sent to the address interface to become the new program counter. The second cycle (figure 4-11(b)) preserves the current processor status register (CPSR) by copying it from the ALU into the SPSR of the exception mode the processor is entering. The final cycle (figure 4-11(c)) saves the return address of the instruction that was about to execute before the exception was entered. For everything except data aborts, this is a modified version of the PC value of the instruction extracted from the end of the PC pipeline as for a normal access to *R15*. However, for data aborts, there is a special holding register that preserves the address of the data transfer instruction that aborted (described in more detail in section 6.1). This value is multiplexed onto the output of the *PC pipe* so that it can be used as the return address of the exception handling routine.



(a) Generate exception vector

(b) Save CPSR ⇒ SPSR

(c) Save return address

Figure 4-11: Exception entry datapath activity

4.4 Instruction flow control

There are three main aspects to instruction flow control in AMULET1:

- Instructions must have access to their correct PC values prior to execution in case it is needed as an operand.
- Instructions which fail their condition tests must be discarded.
- Instructions already prefetched when a branch is taken must also be discarded.

All these areas must be addressed to ensure correct operation and to maintain backwards compatibility with the ARM6.

4.4.1 The ARM PC model and the PC pipeline

In synchronous ARM processors, the value of *R15* - when used as an argument - reflects the instruction prefetching which has been done at the time it is used. In a clocked implementation this is well defined and for most instructions yields a value of the current instruction address + 8 (i.e. PC+8). If the PC is used as a *return address*, i.e when it is moved into the link register (during procedure calls) the value written is PC+4. This is the address of the instruction following the current one and is thus the address of the instruction to return to after the procedure is complete. The value PC+4 can be calculated from PC+8 using the ALU on the way to the link register.

In the synchronous ARM, the PC value is taken directly from the prefetch unit. This is convenient but closely couples the architecture to a specific implementation. In the Micropipelined implementation there is no direct coupling between the prefetch unit and the ALU input. It is therefore not possible to determine how much prefetching has occurred, and hence the relationship between the PC value in the prefetch unit and the required value is not fixed.

For AMULET1 to be code compatible with an ARM6, *R15* must have the correct value; to achieve this each instruction has a PC value associated with it. The instruction and its corresponding PC value synchronize just before execution of the instruction begins. This is implemented by keeping a FIFO (first-in, first-out queue) of PC values and matching them with the instructions as they return from memory. This FIFO is called the “PC pipeline”. Instructions returning from memory are stored in the instruction FIFO prior to execution. Figure 4-12 shows a high-level representation of this operation; this is discussed in more detail in section 6.1.4.

4.4.2 Condition code evaluation

All ARM instructions are conditionally executed depending on the setting of the Current Processor Status Register (CPSR) arithmetic flags, (*N*egative/signed less than, *Z*ero, *C*arry/not borrow/rotate extend, o*V*erflow) and the instruction word bits [31:28]. Thus a comparison of the flag settings and instruction bits [31:28] must be carried out to check whether the instruction should be executed or not. Instructions are typically executed with the ‘always’ condition.

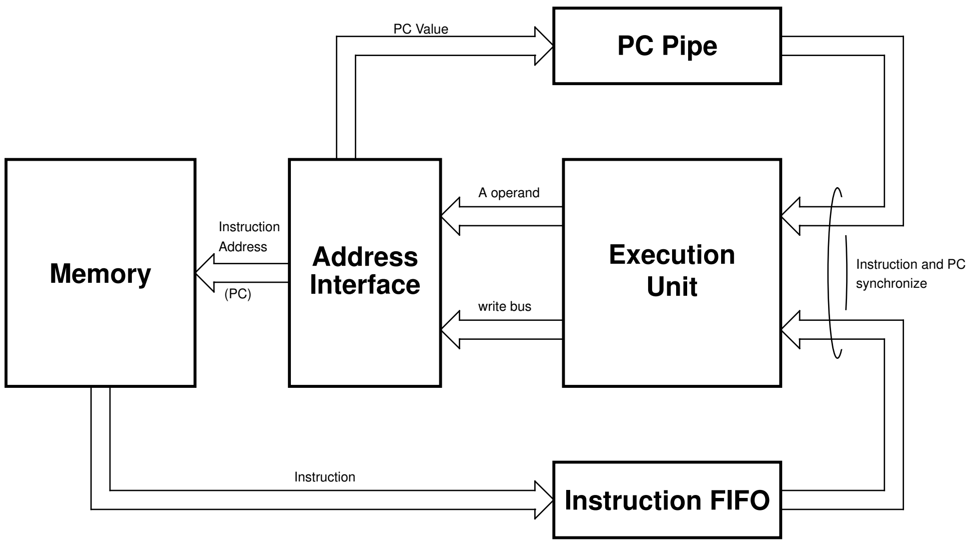


Figure 4-12: Instruction/PC matching

The test of the condition flags is performed in parallel with the second stage of execution (i.e. in parallel with the shift/multiply). A signal is sent from the second stage with the data to the ALU to indicate whether the condition test was successful or not. This is used by the ALU control to invalidate instructions which have failed their test. Instructions that have previously locked a destination register must still return to the register bank to unlock the destination. No data is written back from invalidated instructions; the lock is removed without changing the register contents (see chapter 5 for detail about register locking).

The condition test can be performed in parallel with the shifter because the flags are already valid by this time and cannot change until the ALU is activated. This is a consequence of the dynamic shared latch pipeline structure.

4.4.3 Branch operations

With an autonomous prefetch system, it is not possible to predict how many instructions will have been prefetched when a branch is taken. Prefetched instructions invalidated by the branch must be discarded before instructions from the branch target are processed. This is achieved by maintaining a single bit *colour* flag which changes every time a branch is taken. An instruction has an associated colour bit which indicates the state of the processor colour when the instruction fetch was requested.

When the instruction stream changes, the reference colour changes and hence the colour of instructions subsequently fetched also changes. Each instruction which arrives at the datapath for execution has colour information which indicates whether the instruction was from the original stream (and hence must be discarded) or from the new stream (to be executed). Instructions whose colour differs from the new reference colour are discarded. This continues until instructions from the new stream arrive with the same colour as the reference; these are then executed.

There are two places on the datapath where the instruction colour could be checked: at the ALU or at the primary decode. Each has merits and disadvantages as discussed below.

Colour checking at the ALU

The most obvious position for colour checking is at the ALU, since a mechanism already exists there to discard instructions which fail condition tests; this mechanism could simply be extended to check colour as well. The ALU stage is also the earliest point at which it is known whether an instruction has passed its condition test which determines, in the case of a branch, whether or not the instruction stream will change.

Discarding instructions at the ALU is advantageous if a branch is not taken, because the following instruction will be immediately behind the failed branch in the pipeline. The cost of a branch not taken is the same as an instruction failing its condition test, i.e only 1 cycle.

However, if the branch is taken, instructions to be discarded must still traverse the datapath to have their colours checked at the ALU. The performance of taken branches may be impacted if the time taken to flush the old instruction stream is greater than the branch target reload time. This may be significant if one of the instructions to be discarded is a multiply which can take up to eleven internal cycles in the shift/multiply block (before being discarded at the ALU). However, typically, loading from a non-sequential location is likely to take longer than flushing the datapath.

Another problem with discarding instructions at the ALU is that by the time the instruction reaches the ALU it has already locked its destination registers (if any). Therefore the instruction must still notify the register bank that the destination register should be unlocked, even though its result is to be discarded.

The worst case for both power and performance would be the instruction FIFO being full of prefetched multi-cycle instructions when the instruction stream is changed. Each instruction would enter the datapath, split into multiple cycles, progress down the datapath to the ALU and then be discarded. The activation of the functional units on the way to the ALU would consume power and could take longer than the branch target reload time. Although typical instruction streams may not often exhibit such worst case characteristics, it is still disadvantage of colour checking at the ALU.

Colour checking at the primary decode

The main advantage of checking the colour before any registers are read is that the instruction can then be discarded before entering the datapath. This offers a power saving over the previously described ALU colour checker (no functional units are activated when instructions are being discarded). Another advantage is that the instructions are discarded before entering any computational stages so that even complex instructions can be discarded quickly and efficiently.

As all instructions are conditionally executed, it is not known whether a branch will be taken until its condition codes have been evaluated at the ALU decode. Instructions following the branch cannot be allowed to proceed to the colour checker in the primary decode until confirmation is received whether the branch was taken or not (so that the reference colour can be changed). Confirmation is received when the branch instruction reaches the ALU; the effect of this is that the pipeline stalls at the primary decode waiting for confirmation from the ALU for every possible branch instruction. For taken branches this is very efficient because only instructions which are to be executed from the original instruction stream enter the datapath and hence consume power.

If the branch is not taken (i.e. the branch failed its condition test) then the pipeline between the ALU and the register bank will be emptied for no reason. Investigations into ARM instruction set usage [Jagg89] have revealed that approximately 15-20% of instructions executed are branches, with just over 50% of branches taken. This means that approximately 10% of the instructions executed would stall at the register bank unnecessarily, with the performance penalty that flushing the datapath pipeline carries.

A combined colour checking mechanism

Both schemes described have merit, however each has drawbacks. The solution adopted is a combination of the two. Colour checking is primarily carried out at the ALU decoder. However, when the colour changes, the colour information is transmitted to the primary decode with the aid of an arbiter. Instructions already in progress are discarded at the ALU and instructions waiting in the instruction FIFO are discarded by the primary decode.

Care must be taken to ensure that when instructions are discarded by the primary decode, the corresponding entry in the PC pipeline is also discarded to maintain the correct association between instructions and PC values.

4.5 Exception handling

The ARM architecture specifies four classes of exception that must be handled. These are listed below in order of increasing complexity:

1. Software interrupts and undefined instructions.
2. Instruction prefetch aborts.
3. Hardware interrupts.
4. Data aborts.

For the first class of exceptions, the primary decode simply recognises the bit pattern for the software interrupt (SWI) or undefined instruction and immediately begins the exception entry routine (see section 4.3.7). This is similar to normal instruction execution.

When an instruction prefetch abort occurs the memory responds with a invalid data word marked by an “abort” flag (*Pabt*). This flag is copied into the instruction pipeline along with the invalid instruction where it is queued for entry into the primary decode. When it enters primary decode the prefetch abort flag is detected causing the instruction data to be ignored and the exception entry routine to be entered. The aborted instruction still associates with its PC value, which is read as *R15* by the exception entry routine and stored (after suitable modification) as the exception return address.

4.5.1 Hardware interrupts

The level sensitive hardware interrupts may become active at any time and are completely asynchronous to the processor’s operation. Therefore arbitration is required to prevent any synchronization failure. These interrupts enter the processor between the instruction pipeline and the primary decode (see figure 4-3).

Figure 4-13 shows the arbitration hardware in the event path between the *Ipipe* and the primary decode. Although the interrupts are level sensitive, a change in level which causes the interrupt to become active can be viewed as an event. The transition between levels can be used to trigger a Micropipeline arbiter. This is also a convenient place to manage the reference instruction colour which arrives asynchronously from the ALU when a branch is taken. The robust arbiters described in chapter 3 are used. The operation of the circuit is as follows:

Initially the rising edge of the reset signal (*Cdn*) requests control of all three arbiters in parallel (*r1*). Once all three grants (*g1*) have been received an event is sent to primary decode (*Lreq*) to indicate that the arbiters are locked. Any events arriving on *FIQ, IRQ* or *PCCol* cannot gain control of the arbiters. When the primary decode finishes decoding the first instruction it issues an acknowledge (*Pack*), causing all three arbiters to be temporarily released. If there are any outstanding requests on *FIQ, IRQ* or *PCCol* they then take control of their respective arbiters. The pending event (change in level) is propagated to the arbiter output (*g2*). This output is wired directly back to the “done” input of the arbiter (*d2*) so the arbiter is immediately released again. The net effect of doing this is that the change in level of the input of the arbiter has now propagated to the output. When the next instruction arrives at primary decode it waits for all three arbiters to return to locked (signalled by *Lreq*). The primary decode can then inspect *SFIQ, SIRQ* & *SPCCol* to see if their levels indicate that special action is required. The values are guaranteed to be stable because all the arbiters are locked so any further asynchronous transitions on *FIQ, IRQ* or *PCCol* cannot propagate and affect the primary decode. *SFIQ, SIRQ* and *SPCCol* are stable copies of the inputs which are only allowed to change at a well defined time (between *Pack* and the next instruction arriving), this is shown in figure 4-14. (The external FIQ and IRQ pins are defined to be active Low; the polarity of the interrupt signals shown in the diagrams are active High because an inversion takes place on chip between the pin and arbiters).

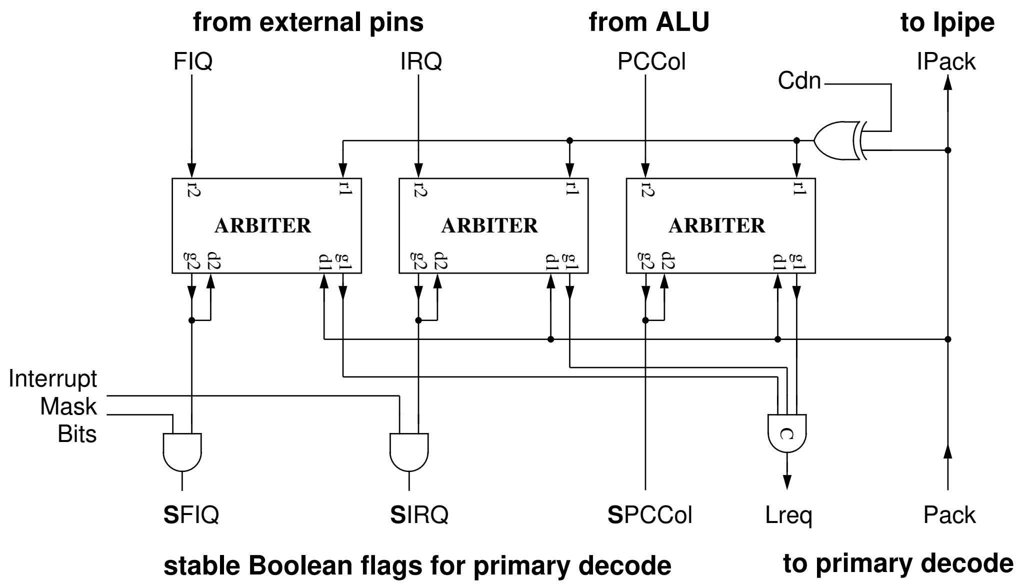


Figure 4-13: Interrupt and PC colour entry into the instruction stream

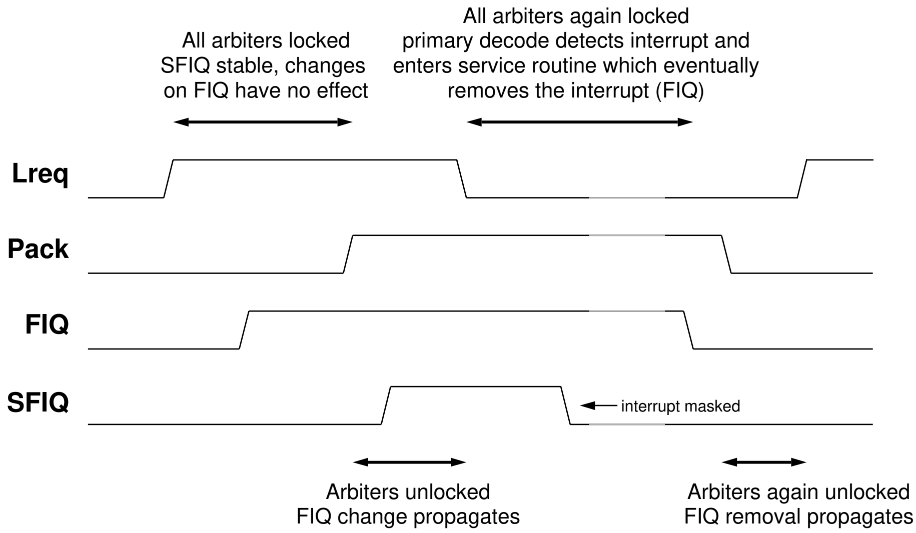


Figure 4-14: Hardware interrupt timing

When the primary decode detects an interrupt request from *FIQ* or *IRQ* the instruction which was about to start decoding is usurped and the exception entry routine is initiated instead. The PC value at the end of the *PC pipe* (and hence the value appearing as *R15*) is the address of the instruction that was usurped (+8). The *R15* value can be read by the exception entry routine and used to form the interrupt return address.

The interrupt flags (*SFIQ, SIRQ*) remain active until they are gated out by writing to the interrupt mask bits in the CPSR. This happens during exception entry (otherwise the processor would repeatedly enter the exception handler).

The exception entry routine inherits the colour of the instruction it usurped. Therefore if the exception entry routine is following a branch which is taken it will be discarded because it has the wrong colour. The interrupt flags remain active so the next instruction which passes the arbiters also gets usurped for exception entry and is also subsequently discarded. This continues until an instruction from the branch target arrives with the correct colour. This time the exception entry routine is successfully executed and the active interrupt flag is masked out.

The reason for not forcing the execution of exception entry cycles which have the wrong colour is that the PC value at the end of the PC pipe associated with the usurped instruction will not give the correct restart address. The correct return address is the branch target. This is automatically delivered by waiting for an instruction with the correct colour.

4.5.2 Data abort overview

Data aborts are by far the most complex exception to handle. The ARM architecture specifies that the instruction which caused the abort must be restartable once the cause of the abort has been removed (e.g. a virtual memory page fault has been fixed). To restart the instruction, the processor needs to determine the address which faulted and the address of the data transfer instruction that issued the faulting address; it must also ensure that its state is preserved whenever an abort is taken (so that a restarted instruction resumes execution in the same environment). An overview of the data abort process is presented here, more detail is provided in the next section.

When a data transfer instruction begins execution, the PC value at the end of the PC pipe is copied into a special FIFO referred to as the exception pipeline (*Xpipe*). When the data transfer reaches the ALU to perform the address calculation, its corresponding PC value is at the end of the *Xpipe*. The ALU sends the data address to memory (via the address interface - see section 4.3.4) and then waits for a response from the memory management unit (MMU) to indicate whether the transfer was successful.

If it was successful, the ALU can continue with the next instruction and the PC value in the *Xpipe* is discarded. If a data abort is signalled then the PC value at the end of the *Xpipe* is copied into a holding latch (*XLatch* - see section 6.1.6) and the exception entry routine entered by signalling to the primary decode. The abort entry routine can then access the exception PC address and store it as the exception return address. To ensure the instruction can restart, the base register (the address of the transfer) must also be preserved. Any base modifications (due to auto-indexing) are allowed to complete because these are deterministic and can be reversed by the exception handling software. Loads which overwrite the base register cannot be permitted to complete because this would destroy the transfer address and hence the instruction could not be restarted.

A single register data load operation, which loads data into the base register, cannot destroy the base if it aborts because no data is retrieved from memory so there is nothing to overwrite the base register with. However, for a load multiple operation it is possible to overwrite the base register early in the sequence and have a subsequent address cause an abort. In this case special measures are needed to ensure that a copy of the base is preserved and is available to restart the instruction (see section 4.3.6).

4.5.3 Data abort signalling

When a data abort is signalled to the ALU, the primary decode must be directed to enter the exception routine. The ALU achieves this by first changing the reference colour so that following instructions are discarded (and therefore do not change the processor state). The change in instruction colour is signalled to the primary decode via the previously described arbiter. This causes all remaining prefetched instructions to be discarded (see figure 4-15) via **SELECT** *S1* and **XOR** *X1*. Once the colour mismatch is established no further instructions can reach the primary decode.

An event is generated when the exception PC holding latch (*XLatch*) becomes full (which must be the result of a data abort). This event is asynchronous with respect to the primary decode and would normally require an arbiter to introduce it into the logic. However, once the PC colour mismatch is detected there can be no further events in the primary decode because incoming instructions are being discarded). The data abort event is therefore safely allowed to pass directly to the primary decode to start the exception entry routine. This safety is ensured because the data abort event is blocked by a transparent latch until the colour mismatch has definitely been established (and hence no more instructions can enter primary decode).

The event to the primary decode is accompanied by a Boolean flag (*Dabt*) to indicate that this is a data abort request. The exception entry routine begins, and saves the exception PC etc. (see section 4.3.7). The various cycles of the data abort entry are forced to execute

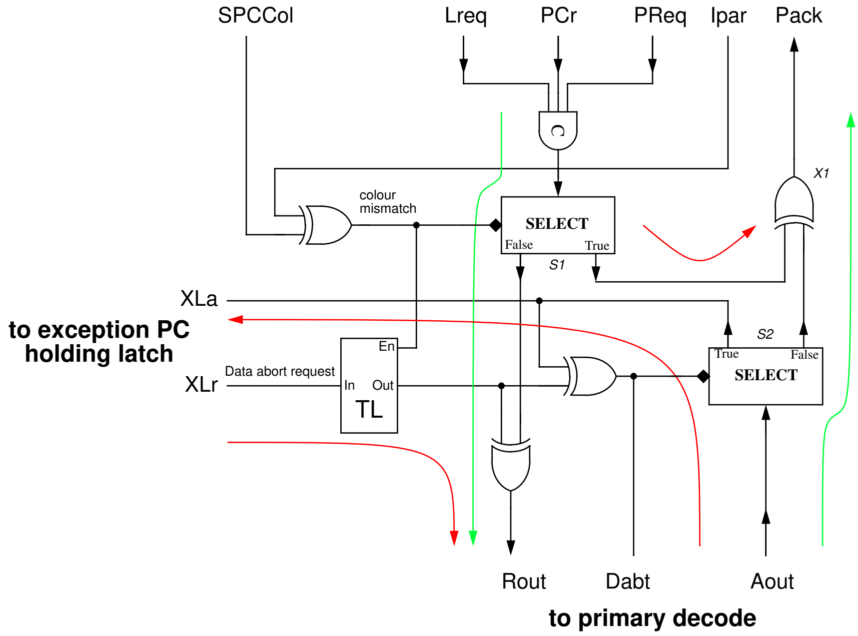


Figure 4-15: Data abort entry into the primary decode

irrespective of the condition codes or colour checking. This is because the data abort entry is not linked to an instruction in the instruction pipeline and does not use an associated PC value; it uses its own value from the *X-Latch.*

When the exception entry is complete the primary decode acknowledges back (*Pack*). This event is steered back to the exception PC holding latch and it also resets the state of SELECT (*S2*) so that it steers subsequent acknowledges back via *X1* to the arbiters and the *Ipipe*. The normal instruction event path is shown in green in figure 4-15, and the data abort event path is shown in red.

Eventually instructions from the exception handler arrive with the correct colour and begin execution as normal instructions.

Chapter 5 : The register bank

A high performance register bank is a central component of a RISC processor. The challenge in an asynchronous design is to maintain coherent register operation while allowing concurrent read and write access with arbitrary timing and dependencies between them. The solution adopted in AMULET1 includes a novel arbiter-free register locking mechanism (described later) which enables efficient read operations in the presence of multiple pending writes.

The special requirements for a register bank in a Micropipelined processor can be specified only after analysing the general operation of a register bank and the hazards which are introduced by a Micropipelined execute unit.

5.1 Register bank operation

A register bank consists of a number of registers with common sets of read and write buses. All registers are the same width; in the 32-bit ARM6 microprocessor they are all 32 bits wide. The microprocessor executes instructions as follows:

The instruction decoder extracts the addresses of the registers to be read (*a, b)* and the register where the result will be written (*w*) from the op-code; these are passed to the register bank to begin execution. The operands are read out and passed to the execution unit where the operation is performed to provide the result. Some time later the result arrives back at the register bank and is written to the appropriate register (*w*). Figure 5-1 shows a

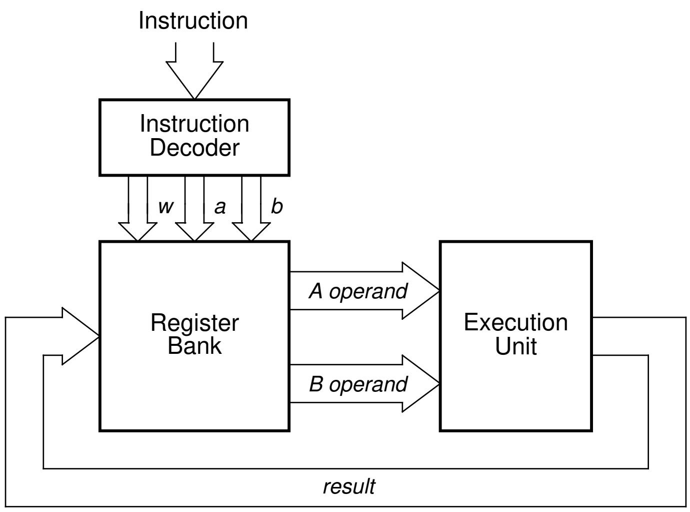


Figure 5-1: Register bank operation

high level view of this process. The number of registers read by any instruction is limited by the number of output ports on the register bank, which in the case of the ARM6 is two.

5.1.1 Internal register structure

Figure 5-2 shows the internal structure of a typical register bank for reading one operand, each register occupies a vertical slice of the register bank. A register comprises of a number of individual memory cells arranged in a vertical stack (32 memory cells for a 32-bit register). Each of the memory cells has an input (which is used for changing its value) and two outputs (which are used to read the value from the cell onto the *A* operand and *B* operand buses). All operations are performed on all the cells of a particular register at the same time. Three enable signals are provided for each register to control reading onto the *A* bus, reading onto the *B* bus and writing from the result bus into the cell.

The *A* bus decoder takes the binary representation of the *A* operand register number (*a*) and converts this into a unary representation (1 out of *N*) used as the *A* bus output-enable for the selected register. Figure 5-2 shows 1 out of *N* registers selected in this way. Similar decoders are provided for the *B* bus and for the write bus which have been omitted from the diagram for clarity.

The *A* operand, *B* operand and write buses all run horizontally through all the registers as shown above; again the *B* and write buses have been omitted for clarity.

At a slightly higher level this can be represented as in figure 5-3. Physically, the decoders are stacked on top of each other with the select wires routed vertically through to the register bank.

5.1.2 Register bank hazards

If the execution phase of the instruction is pipelined then it is possible to have multiple operations in progress at the same time implying that there may be more than one write operation outstanding. To ensure the correct result is written to the appropriate register, a record of destination register addresses must be maintained; this allows a returning result to pair with the correct destination address and to update the correct register.

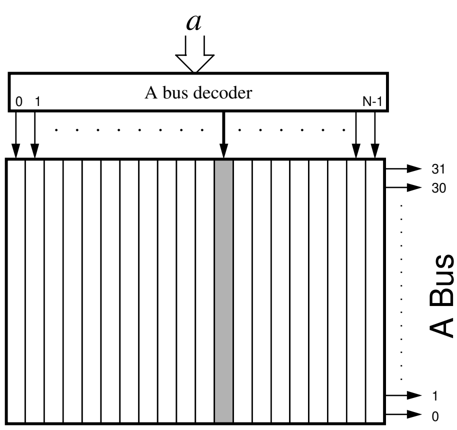


Figure 5-2: Register bank internal structure

Another consequence of a pipelined execution unit is that an instruction may try to read the contents of a register that is to be written by an instruction already in progress. This situation can be caused by an instruction which uses the result of the preceding instruction. If the result has not yet been calculated then the previous value could be read erroneously. To ensure correct operation, the dependencies between instructions must be managed so that reads from registers which have not yet been written wait until the contents are valid.

The third hazard of register bank operation is related to the asynchronous nature of the write operation in relation to a read. The execution unit can take a data dependent time to complete its operation, so there is no fixed time at which a result is expected to return to the register bank. The next read may start as soon as the operand data is latched into the execute pipeline. Therefore reading and writing are asynchronous operations and care must be taken to ensure that any interaction between the two, for example writing and reading the same register, does not cause metastability problems (e.g. when a register value is changed part way through a read operation).

In summary, the three register hazards associated with an asynchronous pipelined processor are as follows:

1. There may be multiple outstanding write operations
2. A read may be requested from a register whose contents are invalid pending a write.
3. Asynchronous read and write operations using the same register may interact unpredictably.

5.2 Write address storage

The obvious way to store the write addresses is in a simple FIFO. Write addresses are put into the FIFO after the instruction reads its operands and removed from the FIFO when the destination address is required to write the result back into the register bank. A standard

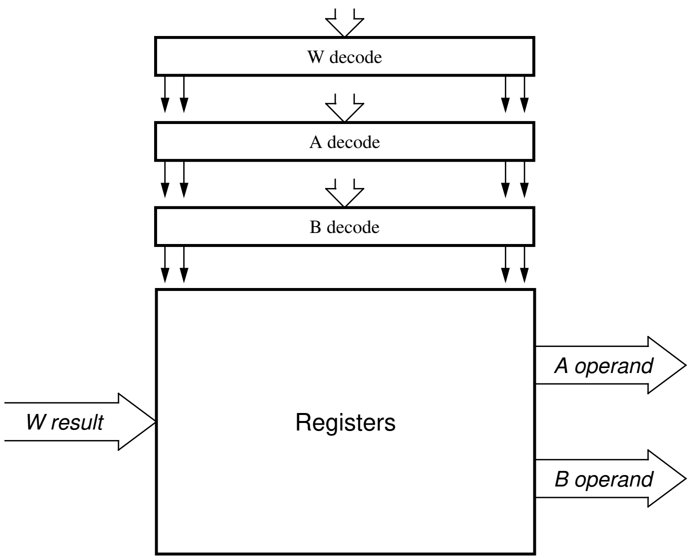


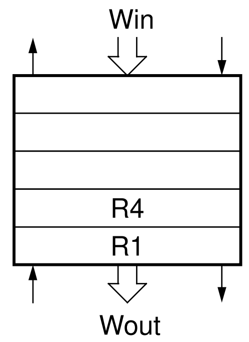
Figure 5-3: Decoder arrangement

1. R1:= R2 + R3

2. R4:= R5 - R6

3. R7:= R9 - R4

(a) Example instruction sequence



(b) State of the FIFO after the issue of instructions 1 & 2

Figure 5-4: Write address storage

Micropipeline provides a simple and efficient way of doing this. Figure 5-4 shows an example instruction stream and how the write addresses are stored within such a FIFO. The example shows that the first two instructions have begun execution and placed their destination write addresses in the FIFO. The third instruction cannot commence until the result of the previous instruction (*R4*) has been written back to the register bank since it is required as one of the operands. The FIFO contains a list of all registers whose contents are invalid. These invalidated registers must be *locked* so that reads to them cannot proceed until the contents are valid. A read operation could therefore inspect the so called “lock FIFO” to determine whether it may proceed or whether it must wait for a value to return. This poses the problem of finding and comparing register values in the asynchronous lock FIFO to determine if the registers to be read are locked. The condition detection mechanism[Pave91] provides a simple and efficient way of achieving this.

5.2.1 An asynchronous register lock FIFO

If the lock FIFO is constructed so that the locked registers are represented as decoded unary values, (i.e. 1 out of N), then each lock entry will have exactly 1 set bit (in the position that corresponds to the register that is locked). For example if there are 32 registers then there will be 32 bits, and if bit 3 is set then this indicates that register 3 is locked pending a write operation.

Using this representation, the determination of whether a register is locked is a matter of establishing whether there is a bit set in a particular column. Figure 5-5 illustrates how the

R7			R4				R0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	1	0	0	0	0
0	0	0	0	0	0	1	0

Figure 5-5: Lock detection

FIFO is interrogated to determine if *R4* is locked and whether the read may proceed or not (the FIFO illustrated contains the unary encoded values of *R1* and *R4*). In this case *R4* is expecting write data so the read must stall.

A further, beneficial, consequence of storing the write address in unary form is that the last stage of the FIFO can be used to drive the write word lines of the register bank without further decoding.

5.2.2 FIFO examination

The columns in the lock FIFO can be examined by simply **OR**ing together successive bits of the FIFO as shown below in figure 5-6. This gives two outputs, one that indicates that the next write operation is to this register and the other indicates that this register has a write pending and is therefore locked. For this simple **OR**ing technique to work in an asynchronous FIFO, three conditions must be met:

1. Empty stages must not interfere with the lock and so must present a zero output.
2. Data must be copied to the next stage before being deleted from the current stage to prevent it from transiently disappearing from the **OR** chain.
3. Data must not be allowed to enter the FIFO while the "*Locked*" output is being examined to prevent asynchronous interaction problems (metastability etc.).

Micropipelines, in general, are transparent when empty, so ensuring empty stages give a zero output is a simple matter of ensuring that the input to the FIFO is held at zero while data is not being entered. Another feature of standard Micropipelines is that they automatically copy data to the next stage before deleting it from the current stage, so the second condition is also satisfied.

Instructions which read and write the same register must not lock the destination until the read is complete otherwise they would stall on their own lock. This requires sequential ordering of the read and lock operations. This high-level control constraint also solves the

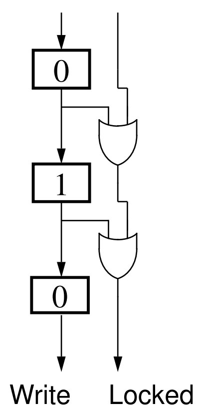


Figure 5-6: Lock interrogation

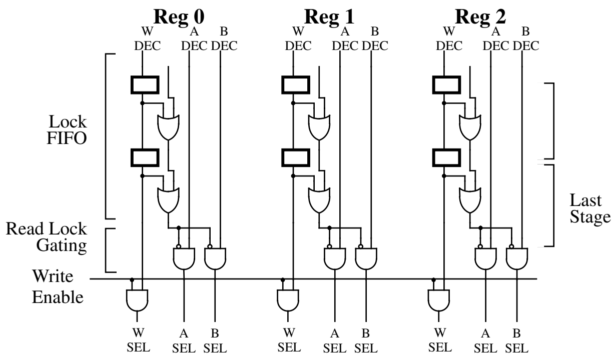


Figure 5-7: Read lock gating

third condition required for correct operation of the lock FIFO. The detail of the high-level control is discussed later.

5.2.3 Stalling reads

The lock FIFO provides a mechanism which indicates that a register has a write pending. This information must be used to prevent any read operations from proceeding until the data has returned and the write operation is complete. Figure 5-7 shows how the lock FIFO *locked* output and the *A* and *B* decode word lines are combined with a simple **AND** gate to stall reads. A lock in any column effectively disables the **AND** gates for that register and so prevents the register read from enabling the contents of the locked register onto the operand buses. A read will remain stalled until the lock entry reaches the last stage and is matched with the corresponding write data. The write operation then completes and the bottom entry of the FIFO is removed; the corresponding set bit will then disappear and allow the stalled read operation to proceed.

The diagram also shows how the last stage of the FIFO is enabled onto the register bank write word lines (*W Sel*) under the control of a write enable signal.

5.3 Asynchronous register bank design

The overall organization of the register bank is shown in figure 5-8 [Pave92a]. The interfaces use the bundled-data convention with transition signalling. Internally, the design employs a combination of two-phase and four-phase techniques, the latter being well matched to the precharge-active cycle of the dynamic circuits used in the basic register cell.

5.3.1 Read and lock operations

A new instruction has its availability signalled by *I-Req*, and presents two register addresses to be read (*a* and *b*) and a register address to be written (*w*) once the execute unit result is available. *I-Req* is stalled until the register bank is ready to start a new read operation when the read decoders are enabled. Concurrently with the read address decoding, the write register address is latched (in *W Latch*).

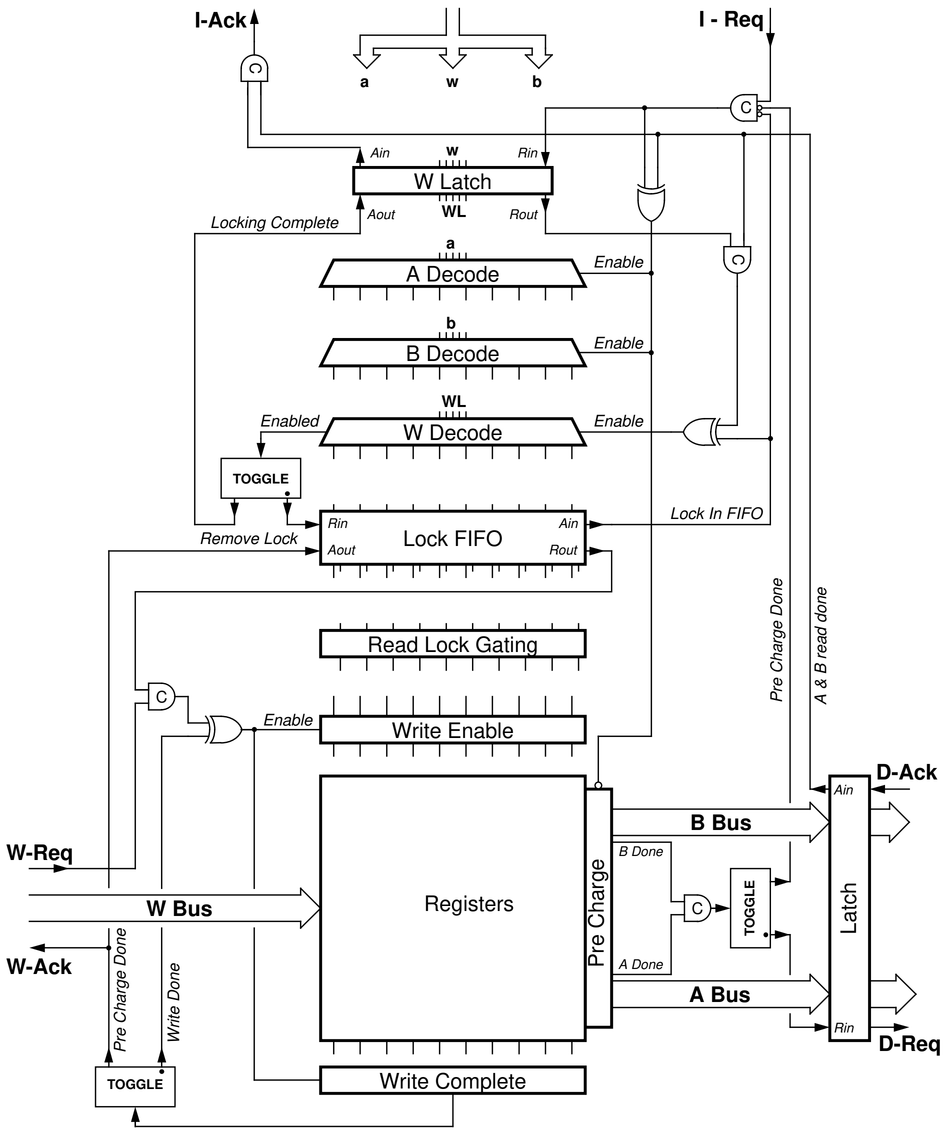


Figure 5-8: Register bank organization

The decoded read addresses present "enables" for the selected registers which are gated with the locked register information. A read of an unlocked register will proceed, whereas a read of a locked register will stall at this point until a write operation clears the register lock.

The register read circuitry uses dynamic techniques to minimise the cell size, with charge retention circuits to give pseudo-static operation. A extra thirty-third bit line gives a matched completion signal and when both register values are available they are latched and passed to the execution path (via the *D-Req* signal) which can begin to process the data with no further delay.

Once the data has been latched, the read decoders are disabled and the read bus precharge is turned on to prepare for the next access. Normally the write address will be latched well

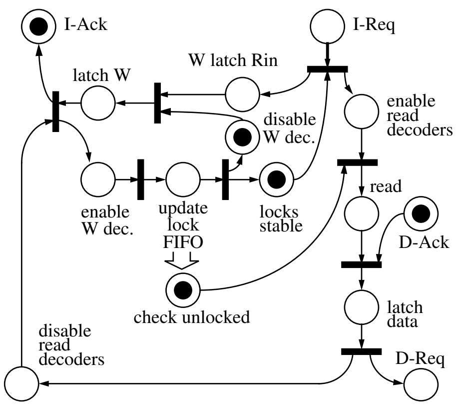


Figure 5-9: A Petri Net model of the read-lock sequencing

before this time, and the instruction acknowledge (*I-Ack*) is issued so that a new instruction can be prepared during the register recovery time.

The write decoder is disabled during the read operation to present inactive inputs to the lock FIFO. Once the read data is latched, the write decoder is enabled and the destination register is locked. As soon as the lock FIFO has accepted the new address, a new instruction may be allowed to start its read operation. The lock logic will continue by disabling the write decoder, it will then free the write address latch for the next value.

The read-lock sequencing is illustrated in Petri Net form [Pete81] in figure 5-9, which shows the critical sequential dependencies in the read operation. Note particularly that the *W* decoder is disabled until the read has completed, in order to ensure that no spurious lock indications are passed via the empty (and therefore transparent) stages in the lock FIFO; similarly it is disabled before the *W latch* is allowed to accept a new value. The next read is allowed to proceed as soon as the locks are stable, since any transient caused by the slow disabling of the *W* decoder will cause at worst a delay in the read operation, never an incorrect action.

The critical path in the register bank (from *I-Req* to *D-Req*) has the minimum number of dependencies on internal operations; this defines the register access latency of the design. The cycle time will include this and the slowest of three independent recovery routes:

- The supply of the next instruction.
- The completion of the locking operation.
- The read bus precharge time
  (omitted from figure 5-9 for clarity).

In general it is expected that the first of the above will be the critical path in determining the register bank cycle time.

5.3.2 Write operations

A write data value (signalled on *W-Req* in figure 5-8) is paired with the decoded write address at the output of the lock FIFO. The appropriate write word line is then enabled and the data written, following which the destination register is unlocked for reading by removing its address from the FIFO. The write operation is self-timed by detecting the transitions on the word line with a wide dynamic OR gate, the same circuit ensures that writes are fully disabled before the write data is allowed to change

5.4 Additional features

There are several features described in this section which were omitted from figure 5-8 to maintain the clarity of the diagram. The write operation includes a Boolean flag to indicate whether the data is valid. This allows instructions which have failed condition tests at the ALU to return to the register bank to remove locks placed previously, without writing any data. The valid flag is used to steer the write request to remove the last item in the lock FIFO without enabling the write word lines. No data is written into the register bank.

The program counter on the ARM is available as a general purpose register (*R15*). The value of the program counter for any particular instruction arrives with the instruction (from the PC pipe) so the register bank simply transfers the value onto the operand bus if required. The multiplexer control is similar to the read control of the general purpose registers; however, as *R15* is not an actual register in the bank, there is no notion of writing to *R15* within the register bank. Therefore there is no lock FIFO entry for *R15* and reads from *R15* can never be stalled on a lock. The value of *R15* is changed by sending the new value to the address interface (a branch operation).

Some instructions do not require all three register addresses (*a*, *b* & *w*) so logic is supplied to bypass a subset of them. To economise on logic for the read operations, instructions which do not need both operands read the PC value instead and then discard it. It is safe to do this because reads from *R15* can never stall.

The register bank also contains several special registers which contain saved versions of the processor status register (*SPSR*); these can be accessed only via the *A* operand bus but are enabled in a similar manner to the general purpose registers. Extra information is provided with the instruction to indicate that the access is to an *SPSR*. Write operations to an *SPSR* can specify that only part of the status word is to be changed (e.g. arithmetic flags, control flags or both), extra information therefore is required for write operations to an *SPSR* to indicate which parts are to change. When a write to an *SPSR* is detected the extra information is stored in a control FIFO and maintained in step with the main lock FIFO; when the write address for the *SPSR* reaches the bottom of the main lock FIFO, the corresponding information in the control FIFO is therefore also at the bottom. This is then used to control the write enable of the two halves of the *SPSR* depending on what the instruction specified.

5.4.1 Dual lock FIFO

Instead of a single lock FIFO as described earlier the implementation has two FIFOs; one for operations that load data from memory and one for internal operations where the data comes from the ALU.

The dual FIFOs allow internal ALU cycles to overtake slower external memory accesses assuming there are no register dependencies between the two. This gives rise to potential compiler code-reordering optimizations to reduce dependencies on loaded data and to increase performance (independent ALU cycles can be executed while the data is being fetched).

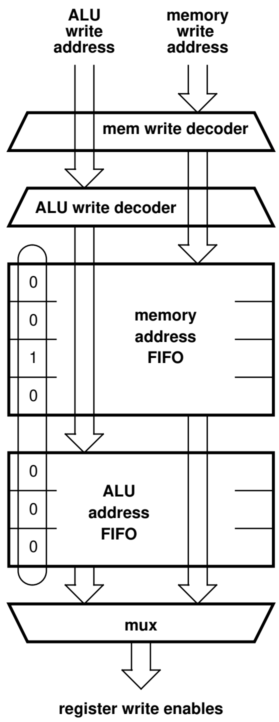


Figure 5-10: Dual lock FIFO configuration

Figure 5-10 shows how the two FIFOs are configured so that they can still correctly indicate a locked condition by **OR**ing across both FIFOs. Two decoders are provided, one for each FIFO and the primary decode supplies two write addresses. This allows an entry to be placed in both FIFOs concurrently and is used to support instructions which have two destination registers (e.g. load data with base writeback; here the two registers are for the data coming back from memory and for the modified base value returning from the ALU). If only one destination address is needed, only one of the write decoders is enabled, and only the appropriate FIFO activated. Once the destination address is in the correct FIFO, the write decode is disabled and the operation proceeds as before.

When a *W-Req* is received it is accompanied by a Boolean flag to indicate whether the data value to be written came from the ALU or memory. This is used to multiplex the correct write destination address onto the write word lines before the write commences. When the write is complete the last entry in the selected FIFO is removed and any read stalled on this write is released automatically.

5.5 Implementation

Storing the write addresses in the lock FIFO in unary encoded form may appear inefficient in the use of silicon area, but it allows the full stack of word control logic (the *A*, *B* and *W* decoders, the lock FIFO, read lock gating and the write enable logic) to be pitch-matched to the register cell block. Figure 5-11 shows the silicon layout of the register bank. The lock FIFO can clearly be seen as the dense regular layout above the datapath register cells. The *A*

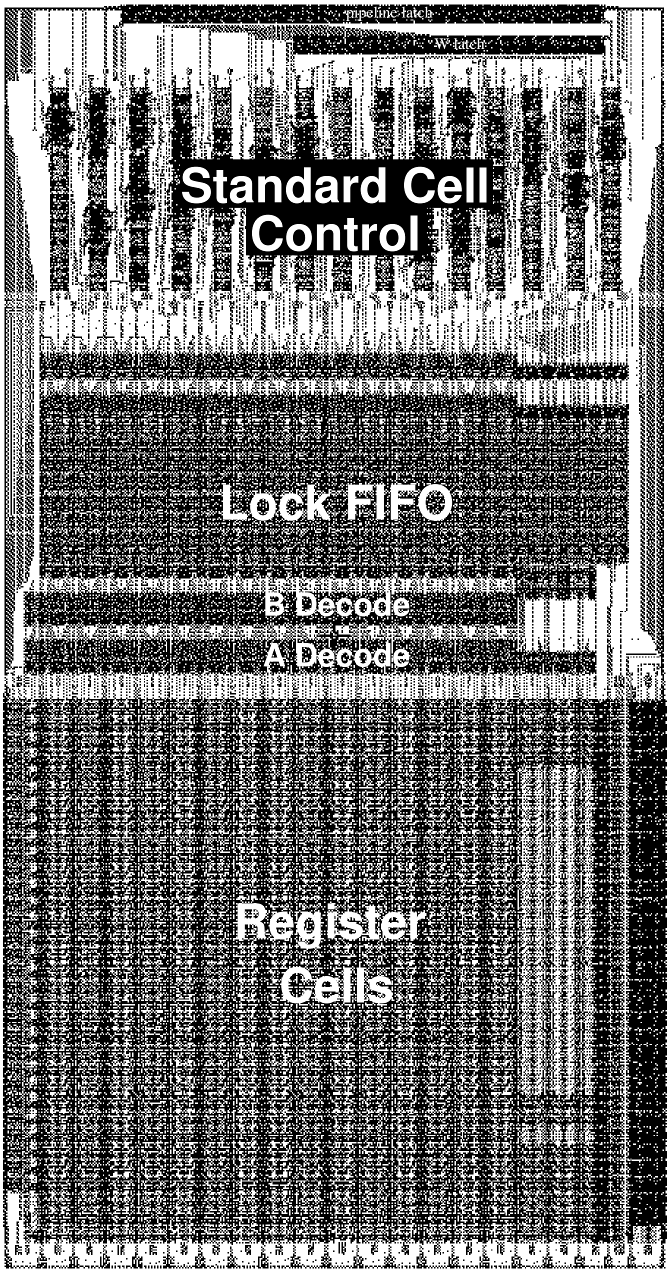


Figure 5-11: Register bank silicon implementation

and *B* decoders (labelled in figure 5-11) below the lock FIFO have the read lock gating built into them with the "locked" signal provided from the FIFO above.

The main core of the bank - the datapath register cells - are identical to those used in the ARM6 with bit 31 at the top and bit 0 towards the bottom. The thirty-third bit used for the self-timing path is in an extra row at the bottom just above the power rails. Using the existing ARM6 register cells maintains the area efficiency of the original synchronous datapath area and illustrates the ability of the Micropipeline design methodology to re-use synchronous elements.

Figure 5-11 also features the 5 *SPSRs* which are the sparse registers to the right of the datapath cells. The sparse nature results from the fact that the processor status word only contains information in bits 31-28, 7, 6 and 4-0 so storage in the other bit positions is not needed (see appendix B for information about the PSR format).

The register bank control logic is implemented in the automatically compiled area of standard cells shown towards the top of the diagram. The decode pipeline latch and the register bank *W latch* can be seen as the two thin horizontal strips at the extreme top of the diagram.

5.6 Future enhancements - register bypassing

Typical instruction streams frequently display the use of the result of one instruction as an operand of the next. Such data dependencies between consecutive instructions can cause a significant reduction in throughput for typical code, compared with best case code without dependencies, if the result is only available to the next instruction after it has been written back to the register bank.

Clocked processors generally use *register bypassing* to allow a result to be re-used without incurring the register write-then-read penalty. The global clock ensures that different parts of the processor are operating at fixed relative times, so the result and operand addresses at two stages can be compared to activate the bypass when appropriate.

In an asynchronous processor there is no such fixed relationship between the timing of operations in different parts of the processor, so explicit synchronisation is necessary if a similar result and operand address comparison is to form the basis of a bypass mechanism. This synchronisation will have a cost in reduced throughput and, since it forces lock-step operation of at least two parts of the processor, is a serious obstacle to fully exploiting the advantages of asynchronous operation.

Two alternatives to register bypassing have been considered which deliver some of the benefits without impeding the asynchronous operation of the pipeline. Although these have not been implemented in the current design, they are included as suggestions for future enhancements and are described below as register through-passing and last result re-use.

5.6.1 Register through-passing

The design shown in figure 5-8 is very conservative in its timing for a write operation which clears a lock and thereby allows a read to proceed. A mechanism under consideration would, with the addition of a latch to the write enable logic, allow the lock to be cleared much earlier in the write process (see figure 5-12). As soon as the write is complete the lock is removed from the FIFO to enable the read to proceed while the write enable lines are held stable by the *write enable latch* until the write enables are safely turned off. This prevents writes to spurious registers.

5.6.2 Last result re-use

Another mechanism under consideration detects data dependencies at the decode stage. Each instruction leaves behind in the instruction decoder a record of its destination register, when the next instruction enters the decoder its operand addresses are compared with this record. If a match is found, the read operation is bypassed and the result is collected for operand use directly. The mechanism has no effect on the design of the register bank in figure 5-8 as it is manifested in additional logic elsewhere in the decode and execution paths.

This second mechanism has better performance than the first when it is applicable, but it has several limitations. A particular problem on the ARM is that all instructions are executed conditionally so an instruction which fails to pass the condition test will not produce a result. By this time its successor may depend on that result, so this mechanism must include logic to determine whether or not an instruction may be annulled (and hence will not produce a result). This adds to the complexity of the design.

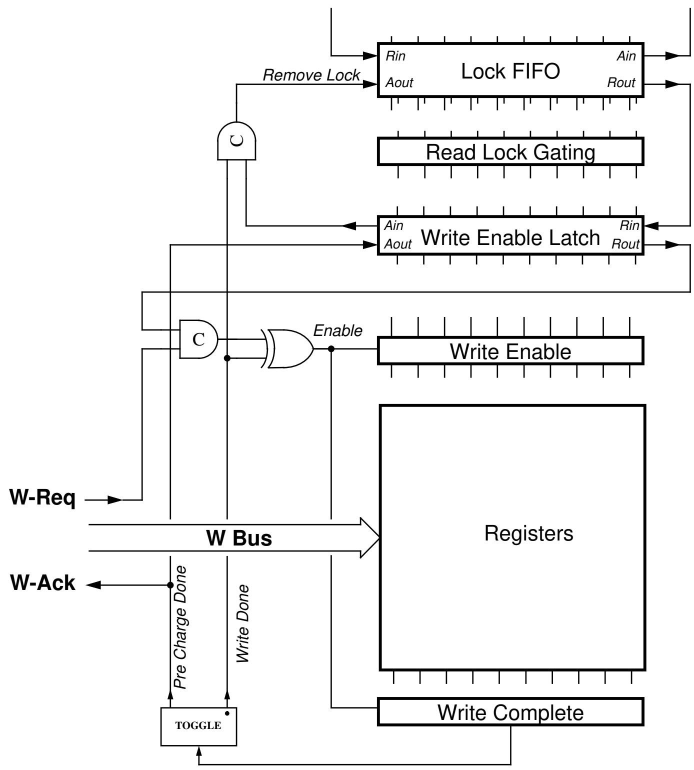


Figure 5-12: Improved register write circuitry

Chapter 6 : Memory interface

The memory interface can be divided into two distinct parts: the address interface, responsible for issuing all address information to memory, and the data interface which manages the flow of data into and out of the processor.

A memory access can be either a read or a write operation. For a write operation, the address supplied by the address interface synchronizes with the data supplied by *data out* before being dispatched to memory. For a read operation, there is no immediate need to synchronize with the data interface when issuing the memory address. However, when the read value returns, the data interface must know whether the value returned is read data destined for the register bank or an instruction to be queued ready for execution.

To enable the data interface to route incoming values to their correct destination, a Micropipeline FIFO containing control information is connected between the address and data interface. For every read request issued by the address interface control information is placed in this *memory control FIFO*. Every read value arriving from memory pairs up with its control information taken out of the memory control FIFO and is steered to the correct destination.

6.1 Address interface

The main function of the address interface is to generate sequential instruction addresses. This is achieved by circulating the PC around a loop containing an incrementer (see section 6.1.7 for further information about the incrementer). For each pass around the loop (shown in figure 6-1(a)) the next sequential address is sent to memory.

After reset, processor execution begins with the memory address register (MAR) being forced to all zeros and an event being generated to start the processor prefetching. The first value sent to memory is *0*. This value is also passed to the incrementer where the next instruction address is generated (PC +4). The incrementer is a dynamic structure so the result is stored in the PC holding registers - *PC HLS* (the reason why two registers are needed is discussed later). If no other address source wishes to use the address interface, then the PC value returns to the MAR via the multiplexers. The next instruction address is dispatched to memory and the process repeats.

The program counter can circulate around this loop at its own speed, decoupled from the actions in the rest of the processor. If the processor wishes to use the address interface to generate the address for a data transfer the PC incrementing loop must be temporarily interrupted; the PC loop is asynchronous to the rest of the processor, so an arbiter is required to manage the interaction. The data transfer begins with the transfer address being supplied on either the write bus or the *A* bus (depending on the instruction and whether pre- or post-indexing is specified - see section 4.3 for details of how addresses arrive at the address

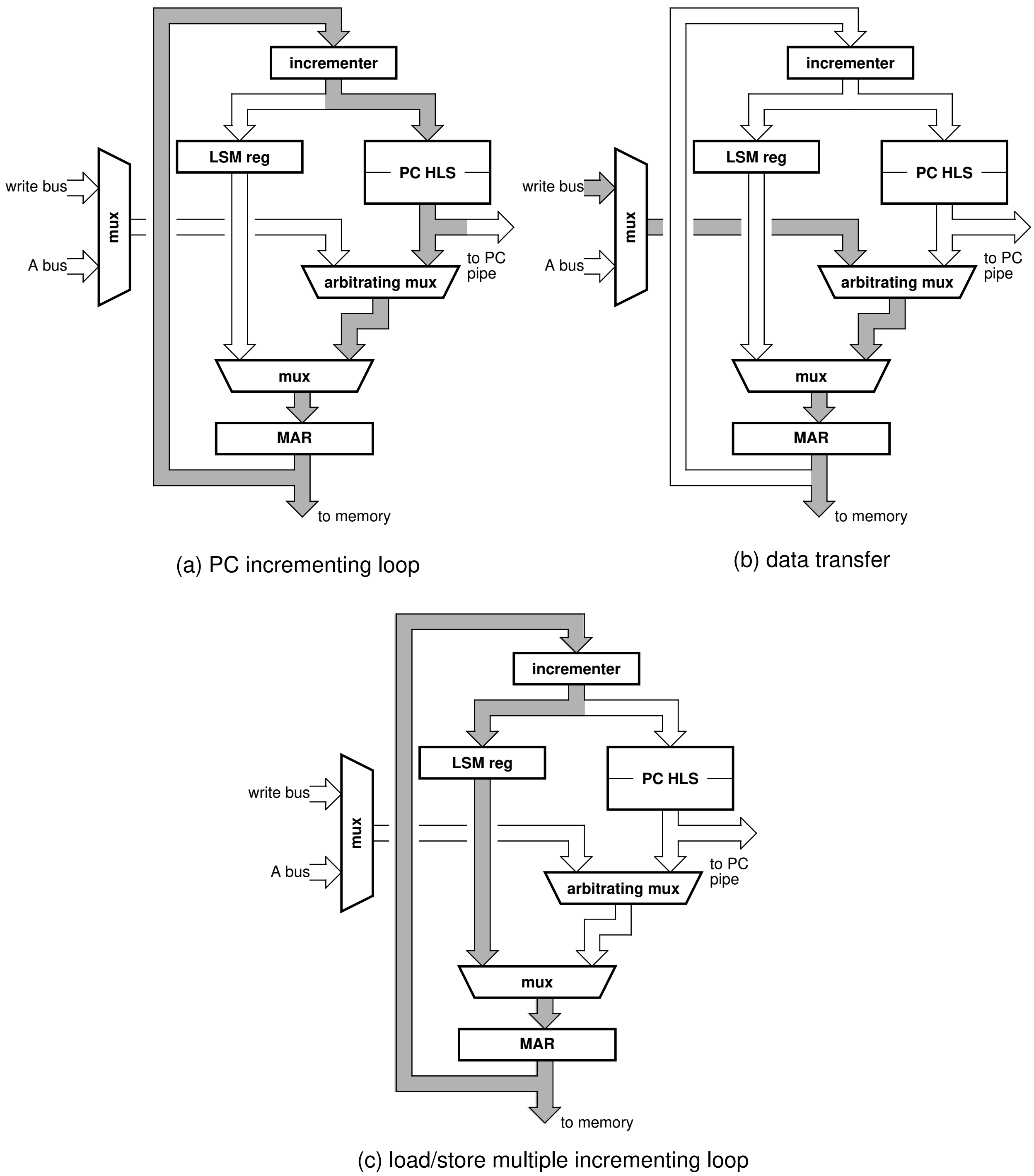


Figure 6-1: The address interface

interface). Arbitration takes place between the PC value and the transfer address to take control of the MAR. Eventually the data transfer takes control and passes its address out to memory via the arbitrated multiplexer and MAR (figure 6-1(b)). Once the transfer is complete the arbiter is released allowing the pending PC to continue to circulate around the loop.

6.1.1 Load/Store multiple operation

A load/store multiple operation sends only the base address of the transfer to the address interface and the sequential transfer addresses are generated by modifying the base using the incrementer (similar to sequential PC address generation). A load/store multiple operation begins with the base address arriving on the write bus in the same way as for a data transfer. Once control is taken of the MAR, the interface waits for a signal from the primary decode. This signal contains information about whether this is the last transfer and for a load operation it indicates whether the data being loaded is the new PC value. The last transfer information is used to stop any further incrementing and if a new PC transfer is signalled then the interface is initialized to expect the new PC value.

Once the signal is received from the primary decode, the address is dispatched to memory; if the transfer is not the last, the address is sent to the incrementer where the next sequential address is generated. The incremented address is stored in the load/store multiple holding register (*LSM reg*).

To take advantage of fast sequential modes of DRAM and some cache memories, the PC is forced to wait until the transfer is complete before prefetching is allowed to continue (this ensures that the LSM addresses are uninterrupted sequential addresses). This is achieved by not releasing the arbiter until the transfer is complete. A consequence of this is that the re-circulating LSM address is not subject to arbitration to take control of the MAR because everything else is locked out of the loop. Therefore, as soon as the *LSM reg* has received the appropriate signal from the primary decode, it can forward its value directly to the MAR as shown in figure 6-1(c). The sequential address generation continues until the primary decode signals that the transfer in progress is the last one. In this case, the address is discarded after the transfer is complete (rather than incrementing it). The arbiter is then released to allow the PC to continue to circulate and issue instruction addresses to memory.

6.1.2 Changing the PC value

The PC value circulating in the address interface has an associated colour related to the main colour mechanism (see section 4.4.3). This is used to discard old circulating PC values when a branch is taken. The address interface control contains a reference colour against which the colour of the circulating PC value is compared; if the colours do not match the PC is removed from the loop and discarded.

When a branch is executed, the new PC value arrives on the write bus from the ALU, bringing with it a copy of the new PC colour. Once the new PC value has taken control of the arbiter, the reference colour is updated to the new value from the ALU. The new PC value then appears on the output of the MAR and is sent to memory and the incrementer.

Once the arbiter is released, the old PC value, which has waited in one of the holding registers, can gain control of the arbiter. As it enters the control circuitry, a colour mismatch is detected and the old PC value is discarded. The new PC value completes circulating the loop and presents a request to the arbiter. This time, when control is granted by the arbiter, the PC colour matches the new reference colour so the PC value is allowed to continue to the MAR unhindered. The PC modification is then complete.

6.1.3 PC loop deadlock

The PC incrementer loop includes two holding latches (*PC HLS* in figure 6-1) to prevent a potential deadlock situation arising as described below:

Consider the case with only one holding latch (*PC HL*); the old PC value is waiting in the holding latch and the new PC already in the MAR as shown in the simplified diagram of the address interface shown in figure 6-2(a).

The new PC value cannot circulate any further until the holding latch is free. In most cases, when the arbiter is released, the old PC immediately gains access and is discarded, freeing up the holding latch and enabling the new PC to continue to circulate.

However, if instead of the old PC value gaining control of the arbiter, a data transfer request takes control, the deadlock situation shown in figure 6-2(b) can occur. Here the data transfer address cannot enter the MAR because the new PC is still occupying it waiting for the PC holding latch to become empty. The holding latch is waiting to gain control of the arbiter and the arbiter will not be freed until the data transfer address is safely in the MAR, hence deadlock results.

If two PC holding latches are provided then the new PC value can move around the loop leaving the MAR clear for any data transfers.

Although data transfer operations immediately following a branch should be discarded because the branch has been taken, a load which is cancelled must still release any destination registers locked previously. To ensure that the write/unlock operations maintain

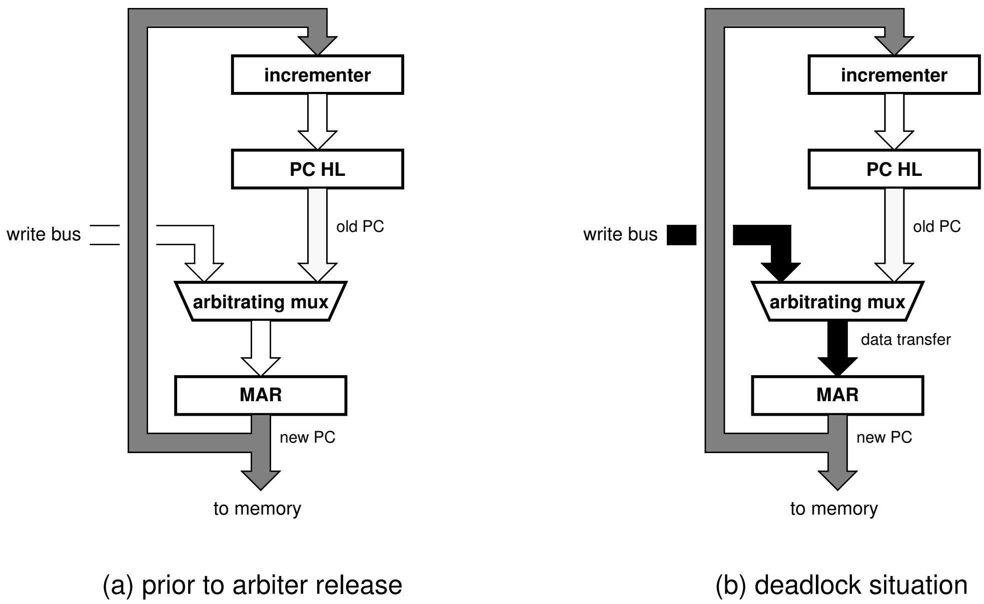


Figure 6-2: PC loop potential deadlock

strict sequential ordering, the unlock request is placed in the memory control FIFO. This is accessed via the address interface so annulled loads still pass through.

When a cancelled load reaches the end of the memory control FIFO in the data interface it is forwarded to the register bank to unlock the destination. Since the load was cancelled, no address was sent to memory and the FIFO entry therefore does not rendezvous with any returning data - it is just sent to the register bank as soon as it reaches the end of the memory control FIFO in the data interface.

The other function the address interface must perform is to supply the PC value to the execution unit for use as the *R15* value. The PC values are stored in a pipeline whose input is connected to the incrementer PC holding latch (as shown in figure 6-1). The operation of the PC pipeline is described below, noting particularly how the value presented as *R15* is controlled to emulate the behaviour of *R15* in the synchronous design.

6.1.4 PC pipeline

As previously described, after hardware reset, the processor starts execution from the bottom of its address space by forcing the MAR to output zero along with the appropriate event control signals. The first value to appear on the output of the incrementer is thus *00000004*. If the output of the incrementer PC holding latch were simply connected to the input of the *PC pipe* then the value PC+4 would be delivered to be synchronized later with the instruction (fetched from address PC). The value actually required for ARM6 backwards compatibility is PC+8. To allow for this, the first PC value after hardware reset is not placed in the *PC pipe*. The first instruction effectively matches up with the “PC+4” of the following instruction hence giving PC+8. This is shown in figure 6-3.

Figure 6-4 shows how the PC values are managed when a branch in the instruction stream occurs; for example, if the instruction at address *4* is an unconditional branch to location *44*. The instructions immediately following the branch (*I3* & *I4*) are assumed to have already been prefetched before the branch instruction was decoded and executed; they must therefore be “thrown away” along with their corresponding PC values. Instruction *I4* is incorrectly matched up with a value from the *PC pipe* (in fact it matches with the PC+4 of the branch target) but this is irrelevant as the instruction is not executed. The branch target instruction is then matched correctly with its PC+8 value.

6.1.5 Instruction overflow deadlock

If no values are removed from the *PC pipe* it will eventually fill; this could happen if the currently executing instruction takes a long time to complete (e.g. a multiplication). When

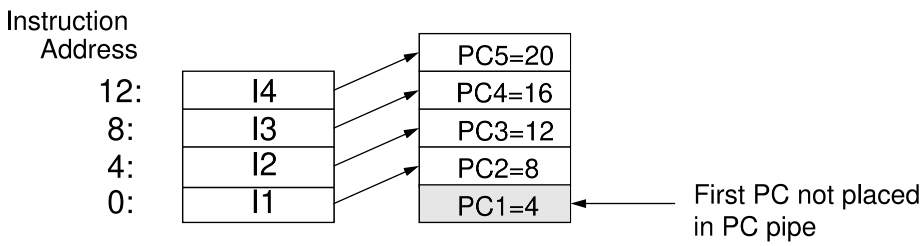


Figure 6-3: Staggering of PC pipe to achieve correct R15 value

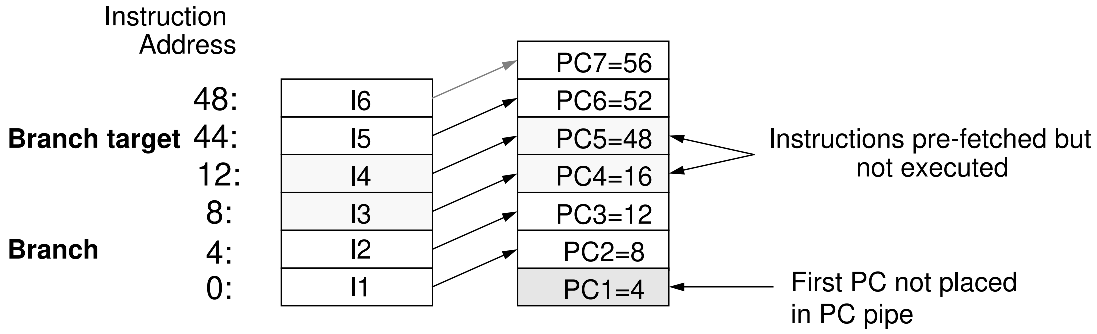


Figure 6-4: PC pipe behaviour across a branch

the *PC pipe* is full, the incrementer PC holding latch will issue a request to it but the *PC pipe* will not acknowledge until there is space in the FIFO for the PC value to be stored. This prevents the incremented PC value reaching the MAR, so no further instruction addresses are issued to memory until the *PC pipe* has some free space.

The length of the PC pipeline determines how many instructions can be outstanding at any particular time, this acts as a self regulating queue or *throttle*. This throttle mechanism is important in preventing a possible deadlock situation as described below.

If the datapath is stalled at the register bank, waiting for a data value to be loaded from memory into a register, instructions will begin to backlog and gradually fill up the instruction FIFO. If there was no throttling mechanism it would be possible for the instructions to backlog all the way into the memory. If the data value happens to be behind this backlog, it would not then be able to complete the transfer to the register bank because the instructions are blocking the memory interface. The instructions are backlogged because of the stall in the register bank which cannot be released because the data value cannot get to the register bank; hence the system is deadlocked. This is shown in figure 6-5.

To ensure that this deadlock can not happen the number of instruction requests issued to memory must not exceed the number of spaces in the instruction FIFO. Thus any backlog is restricted internally to the instruction FIFO itself, leaving the data interface clear to allow the data value to return.

PC pipeline Length

The address interface stops issuing instruction addresses when the *PC pipe* becomes full, this is a function of the number of stages in the *PC pipe*. If the pipe is N stages long then when it is full, N+1 instruction address will have been issued to memory (the first PC value after reset does not enter the *PC pipe*). Therefore the instruction FIFO must be at least N+1 stages long to accommodate all instructions returning from memory, associated with values in the PC pipe. When the *PC pipe* is full, there will also be a PC value in the PC holding latches waiting to enter the *PC pipe*. The instruction corresponding to the PC value in the holding latch will have already been sent to memory, so there are in fact N+2 outstanding instructions which need accommodation in the instruction FIFO (*I pipe*) to prevent deadlock.

If a branch is taken when the old PC value is waiting to enter the *PC pipe* (with N+2 instructions already outstanding) the new PC value arrives from the ALU and is sent out to

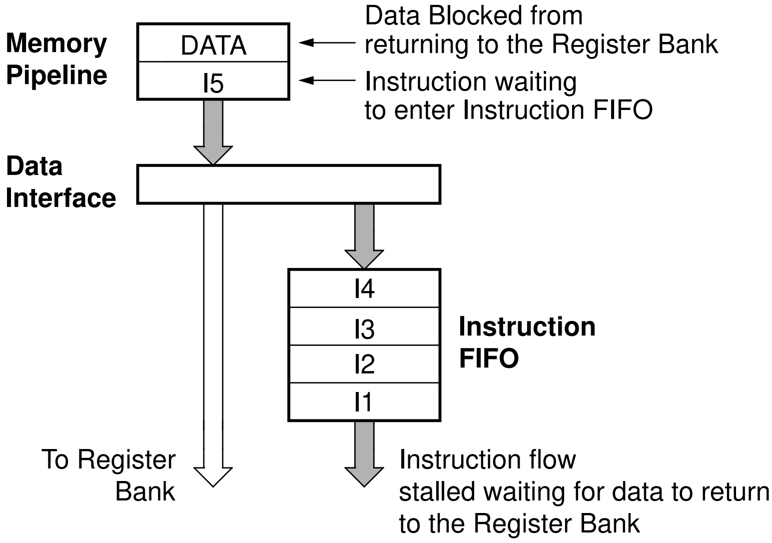


Figure 6-5: Potential processor deadlock

memory making the total of outstanding instructions N+3. If a load follows the branch then this is annulled, but must still unlock any destination registers locked previously. To ensure that the write/unlock operations maintain strict sequential ordering, the unlock request is placed in the memory control FIFO. This is accessed via the address interface so annulled loads still pass through.

If the processor is stalled waiting for the destination of the cancelled load to be unlocked, deadlock will ensue if the memory control FIFO is blocked by any outstanding instruction requests, i.e. all outstanding instructions must be in the instruction FIFO. The *IPipe* must therefore be 3 stages longer (N+3 stages) than the *PC pipe* (N stages). This demonstrates the complex relationship between the depth of the PC and instruction pipelines.

6.1.6 PCpipe implementation

The PC pipelines are split into two parts: PC storage for supplying *R15* values (*PC pipe* in figure 6-6) and *Xpipe* for storing the addresses of instructions in progress which could potentially generate a data abort (i.e. all data transfer instructions).

When an instruction begins execution its corresponding PC value is in the last stage of the *PC pipe*; for normal operation this value is selected by the multiplexer and made available as the *R15* value needed by the register bank. Once the primary decode of the instruction has completed, the PC value can be released, if the instruction is not a data transfer the PC value is removed from the *PC pipe* and discarded. If however the instruction is a data transfer then the PC value is placed in the top of the *Xpipe*, and is removed from the bottom only when the corresponding memory access is complete.

Figure 6-7 shows the detailed operation for removing a PC value from the *Xpipe*. Each data transfer has a corresponding PC value in the *Xpipe* which awaits a response from the memory management unit (MMU). There is a single PC value and MMU response for each load/store multiple transfer.

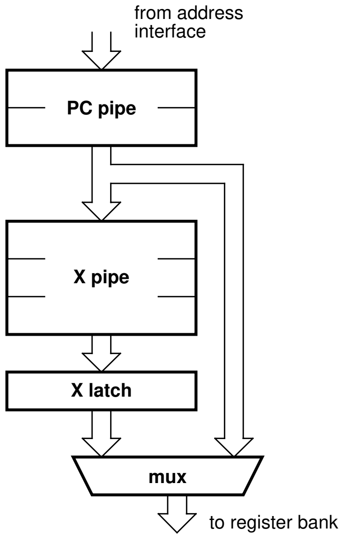


Figure 6-6: The PC pipelines

When the PC value reaches the last stage of the *Xpipe* its request out signal primes a **Decision-Wait** element ready for the MMU response. The MMU issues an event on <u>either</u> *data abort* <u>or</u> *no data abort*. If there is no data abort then the request out of the *Xpipe* is steered back as its own acknowledge (via the **XOR**), effectively removing the PC value in the process.

When a data abort is signalled the last stage of the *Xpipe* (containing the PC+8 of the instruction that aborted) is copied into the exception holding latch (*X-Latch*). The request out of the holding latch is the event used to indicate to the primary decode that a data abort has occurred. The exception entry routine needs access to the address of the failed instruction so that it can retry the instruction when the cause of the exception has been

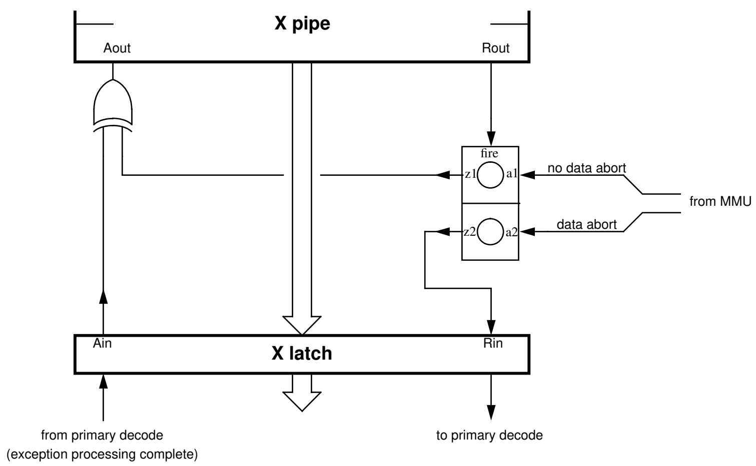


Figure 6-7: Data abort PC storage

removed. The exception address (in *X-Latch*) is multiplexed onto the output of the PC pipelines and is made to appear as *R15* in the register bank, where it can be read and copied into the exception return register, so that when the data abort recovery software routine exits, the processor automatically returns and retries the aborted instruction. Once the exception PC is safely stored elsewhere, the *X-Latch* can be acknowledged and returned to its empty state.

6.1.7 Incrementer

The incrementer is constructed using a simple ripple-carry mechanism with completion detection. Each bit-cell is simple and identical as shown in figure 6-8. The worst case time for this style of incrementer is large because the carry may have to propagate through 30 bits (since PC and LSM are always word aligned there is no need to consider the bottom 2 bits, hence 30 bits worst case). On average, two bits change per operation [Gars92], so the typical case is much faster than the worst. In a synchronous system the cycle time is limited by the worst case, but with an asynchronous implementation the cycle time can vary depending on the data and the extent of carry propagation, so the circuit can be optimized for typical values instead of worst case.

The incrementing operation begins with a carry injected into the lowest bit; if the input to the stage is Low (0) the carry will propagate no further (0+1=1 carry 0). The input selects the path the carry takes through the de-multiplexer; with the input Low (0), the carry is steered to the completion detection circuitry because the increment is finished; if the input is High (1), the carry-in must propagate through the stage when it arrives, to the next highest bit (1+1= 0 carry 1). This is done by the input selecting the other output of the de-multiplexer. The carry continues to propagate until it reaches a stage where the input is Low; here, instead of the carry propagating any further, it is used to signal competition.

The completion circuitry is implemented by a simple wired-OR of the completion signal in every bit. Only one bit will signal completion per incrementing operation. The circuit is implemented using dynamic circuitry to reduce the transistor count to only 20 transistors per bit. Weak-feedback charge retention is provided on the carry inputs to ensure that the

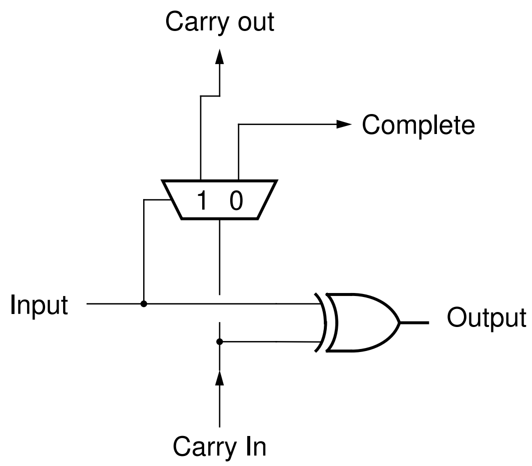


Figure 6-8: Simple ripple-carry incrementer cell

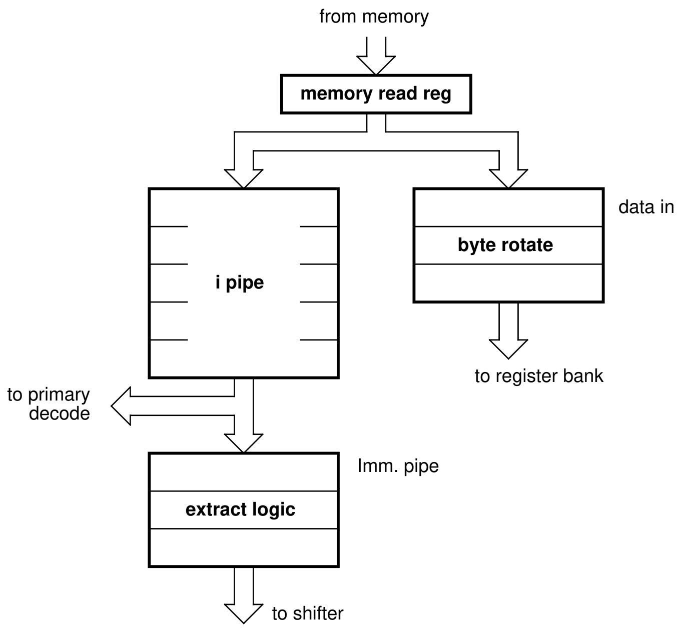


Figure 6-9: Data-in organization

carry inputs of the higher order bits do not decay after the incrementing has completed and hence cause power dissipation.

6.2 Data interface

The data interface is much simpler than the address interface. The *data out* section is simply a two stage FIFO with optional byte replication built into the input of the second stage. The *data in* logic is more complex.

6.2.1 Data in

Values arriving from memory are initially stored in the memory read register (see figure 6-9). For each value, the data interface extracts the corresponding control information from the memory control pipe, this determines whether the value is an instruction or data value.

The instruction pipeline (*Ipipe*) is a simple 5 stage Micropipeline configured as fast forward latches (see figure 3-26) to reduce latency through the *Ipipe*. The end of the instruction FIFO is connected to the primary decode and the immediate field extractor (*Imm. pipe*).

The immediate extractor consists of two Micropipeline stages with intermediate extraction logic. It is possible to determine the size of immediate value to extract (8, 12 or 24 bits) by examining just two bits of the instruction word, so the extraction logic is quite simple. The input stage is configured to be a normally closed (blocking) latch so that, when the immediate extractor is not in use, transitions on its input caused by passing instructions do not cause internal power to be dissipated. The output of the immediate extractor is connected to a multiplexer on the input of the shifter.

The final part of the data interface is the logic which processes data destined for the register bank (*data in*). This consists of two Micropipeline stages with logic to perform rotates in byte quantities for non word-aligned loads and gating to mask out the top 24 bits for byte reads. It also contains logic that can be used to alter which byte in the word is addressed as byte zero (i.e. it can change from “little-endian” to “big-endian” operation).

As the majority of values returning from memory are instructions, the data processing part has a blocking latch on its input to prevent internal power dissipation when it is not needed (i.e. when the incoming value is an instruction).