Lecture 02

An Introduction to Processor Design (2)

1.4 Instruction Set Design

- If the MU0 instruction set is not a good choice for a high-performance processor, what other choice are there?

- 4-address instructions

f bits	n bits	n bits	n bits	n bits
function	op 1 addr .	op 2 addr .	dest. addr .	next_i addr .

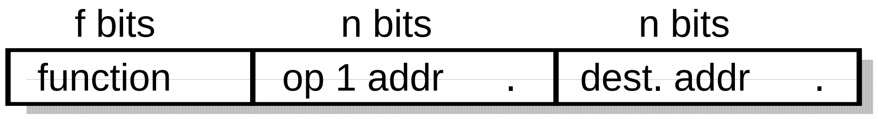
ADD d, s1, s2, next_i; (d:=s1+s2)

- 3-address instructions (e.g., ARM IS)

f bits	n bits	n bits	n bits
function	op 1 addr .	op 2 addr .	dest. addr .

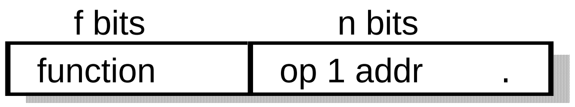
ADD d, s1, s2; (d:=s1+s2)

- 2-address instructions (e.g., Thumb IS)



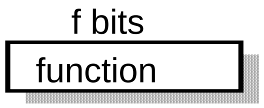
ADD d, s1; (d:=d+s1)

- 1-address instructions (e.g., MU0)



ADD s1; (ACC:=ACC+s1)

- 0-address instructions (e.g., Inmos Transputer)



ADD ; (top_of_stack:=top_of_stack+next_on_stack)

Instruction Types

- A general-purpose instruction set can be expected to include instructions in the following categories:
  - **Data processing**: e.g., add, subtract, multiply
  - **Data movement**: that copy data between memory and memory, or between memory and register.
  - **Control flow instructions**: that switch execution from one part of the program to another, possibly depending on data values.
  - **Special instructions to control the processor's state**: for instance to switch into a privileged mode to carry out an OS function.

- Orthogonal:

  An instruction set is said to be orthogonal if each choice in the building of an instruction is independent of the other choices. An orthogonal instruction set is easier for the assembly language programmer to learn and easier for the complier writer to target. The hardware implementation will usually be more efficient too.

Addressing Modes (1)

- When accessing an operand for a data processing or movement instruction, there are several standard techniques used specify the desired location.
  - Immediate
    - The desired value is presented as a binary value in the instruction.
  - Absolute
    - The instruction contains the full binary address of the desired value in memory.
  - Indirect
    - The instruction contains the binary address of a memory location that contains the binary address of the desired value.
  - Register
    - The desired value is in a register, and the instruction contains the register number.
  - Register indirect
    - The instruction contains the number of a register which contains the address of the value in memory.

Addressing Modes (2)

- When accessing an operand for a data processing or movement instruction, there are several standard techniques used specify the desired location.

  - Base + offset
    - The instruction specifies a register (the base) and a binary offset to be added to the base to form the memory address.
  - Base + index
    - The instruction specifies a base register and another register (the index) which is added to the base to form the memory address.
  - Base + scaled index
    - As above, but the index is multiplied by a constant (usually the size of the data item, and usually a power of two) before being added to the base.
  - Stack
    - An implicit or specified register (the stack pointer) points to an area of memory (the stack) where data items are written (pushed) or read (popped) on a LIFO basis.

Control Flow Instructions

- A control flow instruction is used to modify the PC explicitly.
  - Conditional Branches
    - Branch if a particular register is zero
    - Branch if two specified registers are equal
  - Condition Code Register
    - A special-purpose register to control the conditional branches
  - Subroutine Calls
    - The return address could be placed in a register, memory, or pushed onto a stack.
  - Subroutine Return
  - System Calls
  - Exceptions

1.5 Processor Design Trade-offs

- CISC vs. RISC
- What processors do
  - How to make them go faster
    - Pipelining
    - Cache memory

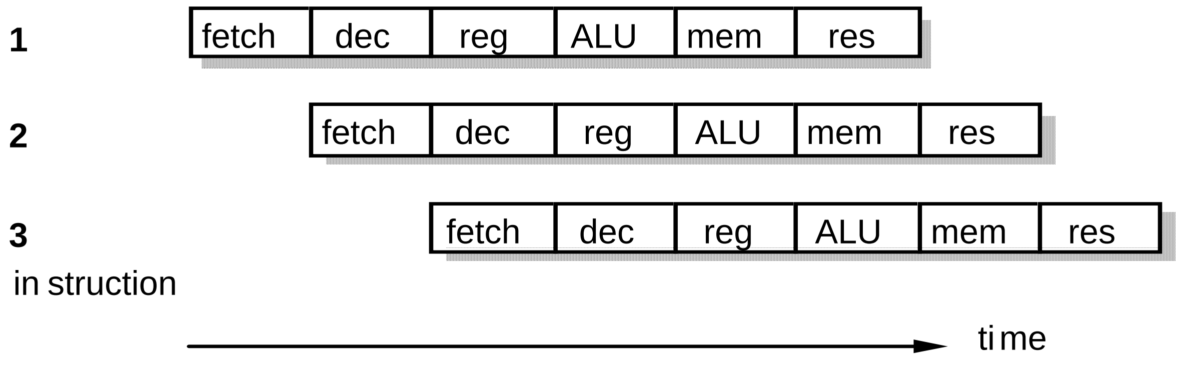
Instruction type	Dynamic usage
Data movement	43%
Control flow	23%
Arithmetic operations	15%
Comparisons	13%
Logical operations	5%
Other	1%

Typical dynamic instruction usage

Pipelines

- A processor executes an individual instruction in a sequence of steps. A typical sequence might be:
  - Fetch (fetch)
  - Decode (dec)
  - Access operands from register (reg)
  - Combine the operands to form the result or a memory addr. (ALU)
  - Access memory for a data operand, if necessary (mem)
  - Write the result back to the register bank (res)

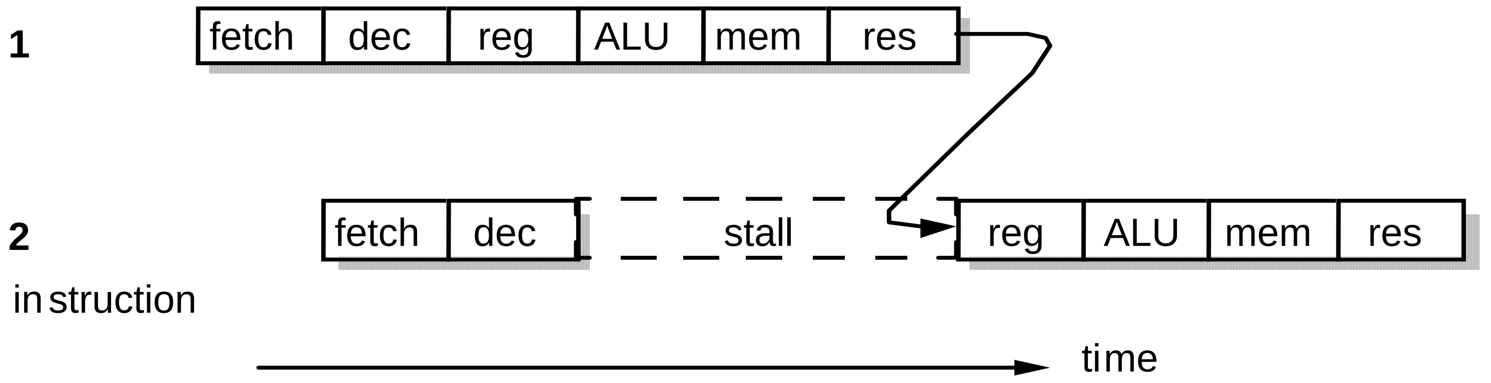
- An obvious way to improve the utilization of the hardware resources, and also the processor throughput, would be to start the next instr. before the current one has finished. This technique is called ***pipelining***, and is a very effective way of exploiting concurrency in a general-purpose processor.



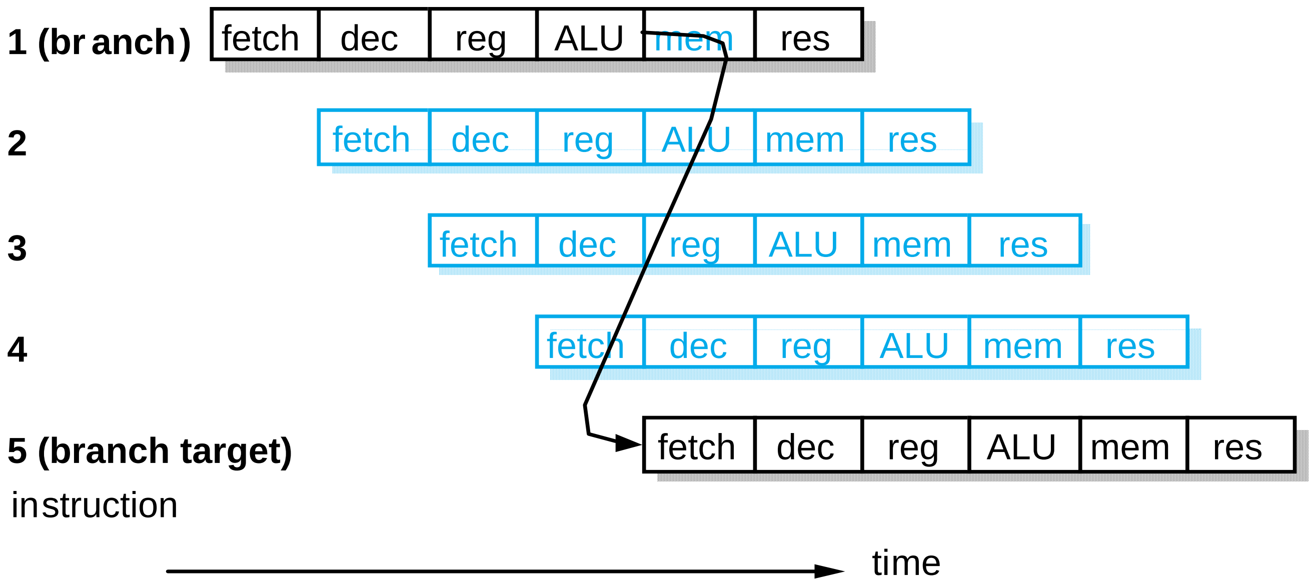
Pipeline Hazards

- Pipeline hazards
  - Write-after-read
  - Write-after-write
  - **Read-after-write**
  - **Control hazard**

- Read-after-write pipeline hazard



- Pipelined branch behavior
  - **Control hazard**
    - If the branch target calculation is performed in the ALU stage, 3 instructions will have been fetched.
    - It is better to compute the branch target earlier if possible, even though this will probably require dedicated hardware.
    - If branch instruction have a fixed format, the target may be computed speculatively during the "dec" stage.


1 (branch)
fetch dec reg ALU mem res
2
fetch dec reg ALU mem res
3
fetch dec reg ALU mem res
4
fetch dec reg ALU mem res
5 (branch target)
fetch dec reg ALU mem res
instruction
time

1.6 RISC vs. CISC

- In 1980 Patterson and Ditzel published a paper entitled '*The Case for the Reduced Instruction Set Computer*', in which they expounded the view that the optimal architecture for a single-chip need not be the same as the optimal architecture for a multi-chip processor.

- Their argument was subsequently supported by the results of a processor design project undertaken by a postgraduate class at Berkeley which incorporated a Reduced Instruction Set Computer (RISC) architecture. This design Berkeley RISC I, was much simpler than the commercial CISC processors of the day and had taken an order of magnitude less design effort to develop, but delivered a very similar performance.

RISC Architecture

- A fixed (32-bit) instruction size with few formats; CISC processors typically had variable length instruction sets with many formats.
- A load-store architecture where instr. that process data operate only on registers and are separate from instr. that access memory; CISC processors typically allowed values in memory to be used as operands in data processing instructions
- A large register bank, all of which could be used for any purpose, to allow the load-store architecture to operate efficiently; …

RISC Organization

- Hard-wired instruction decode logic; CISC processors used large microcode ROMs to decode instr.
- Pipelined execution; CISC processors allowed little, if any, overlap between consecutive instr.
- Single-cycle execution; CISC processors typically took many clock cycles to complete a single instr.

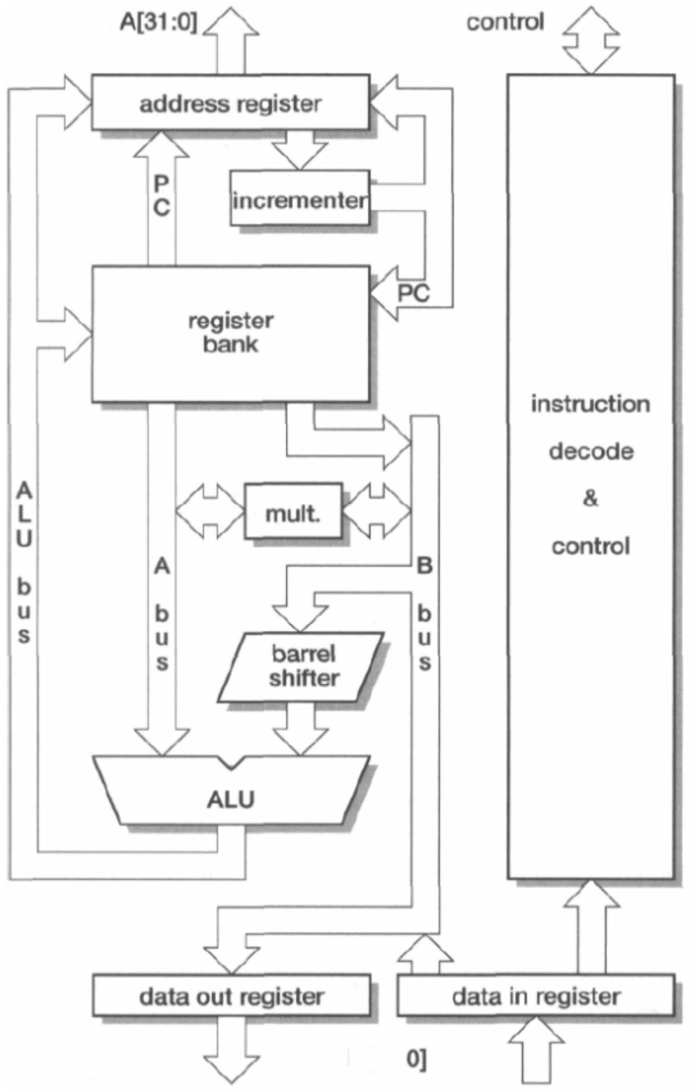
- RISC advantages
  - A smaller die size
  - A shorter development time
  - A higher performance

- RISC drawbacks
  - RISCs generally have poor code density compared with CISCs.
  - RISCs don’t execute x86 code.

ARM Organization and Implementation

3-Stage Pipeline ARM Organization

- The principal components of an ARM organization with a 3-stage pipeline are:
  - Register bank:
    - 2 read ports & 1 write ports
    - additional read and write ports for accessing r15
  - Barrel shifter:
    - Shift or rotate one operand by any number of bits
  - ALU:
  - Address register and incrementer:
  - Data register:
    - Hold data passing to and from mem.
  - Instruction decoder and associated control logic:



The 3-stage Pipeline

- ARM processor up to the ARM7 employ a simple 3-stage pipeline with the following pipeline stages

  - Fetch
    - The instruction is fetched from memory and placed in the instruction pipeline
  - Decode
    - The instruction is decoded and the datapath control signals prepared for the next cycle. In this stage the instruction ‘owns’ the decode logic but not the datapath.
  - Execute
    - The instruction ‘owns’ the datapath; the register bank is read, and operand shifted, the ALU result generated and written back into a destination register.

- When the processor is executing simple data processing instructions the pipeline enables one instruction to be completed every clock cycle. An individual instruction take three clock cycles to complete, so it has a three-cycle latency, but the throughput is one instruction per cycle.

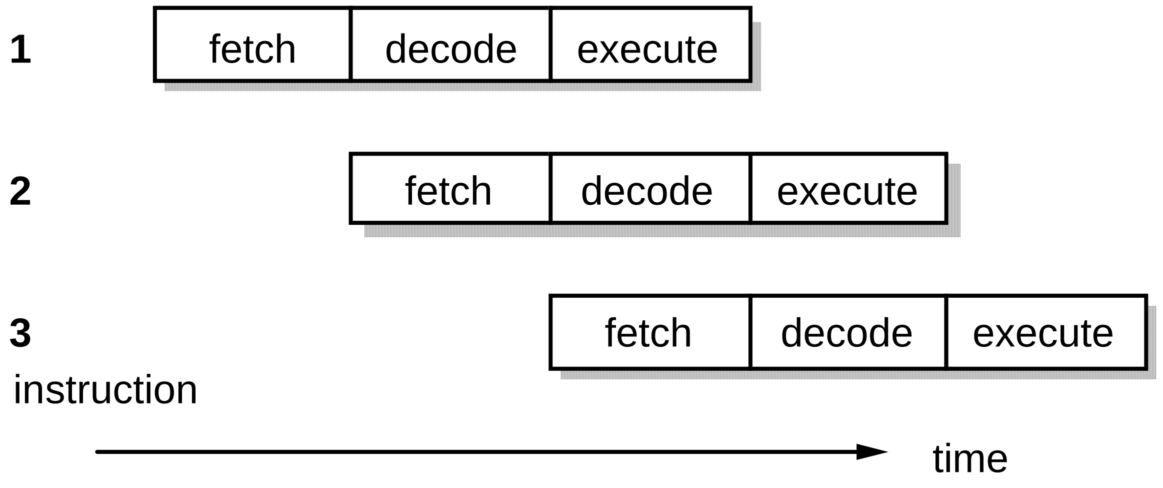


Figure 4.2. ARM single-cycle instruction 3-stage pipeline operation.

- When a multi-cycle instruction is executed the flow is less regular, as illustrated in Fig. 4.3. This shows a sequence of single-cycle ADD instructions with a data store instruction, STR, occurring after the first ADD. The decode logic is always generating the control signals for the datapath to use in the next cycle, so in addition to the explicit decode cycle it also generating the control for the data transfer during the address calculation cycle of the STR.

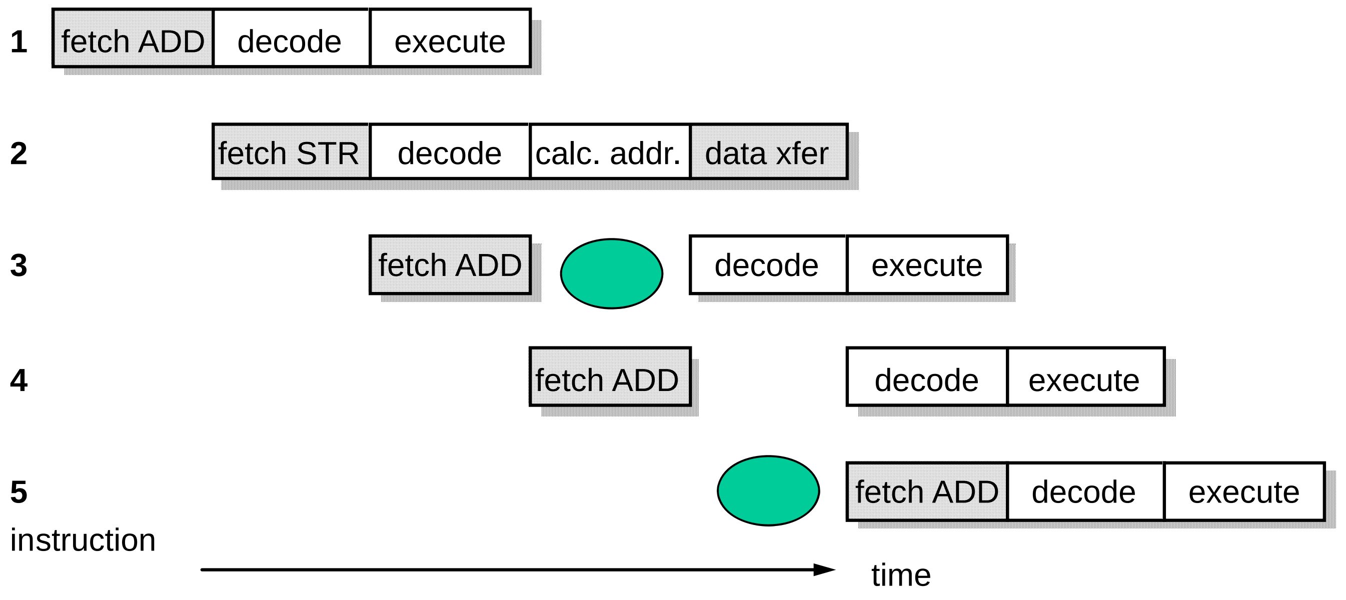


Figure 4.3. ARM multi-cycle instruction 3-stage pipeline operation.

5-Stage Pipeline ARM Organization

- The 3-stage pipeline used in the ARM cores up to the ARM7 is very cost-effective, but higher performance requires the processor organization to be redesigned. The time, T_{prog}, required to execute a given program is given by:

$$T_{prog} = (N_{inst} \text{ X } CPI) / f_{clk}$$

- Since N_{inst} is constant for a given program, there are only two ways to increase performance:
  - Increase the clock rate, f_{clk}.
    - This requires the logic in each pipeline to be simplified, therefore, the number of pipeline stages to be increased.
  - Reduce the average number of clock cycle per instruction, CPI.
    - This requires either that instructions which occupy more than one pipeline slot in a 3-stage pipeline ARM are re-implemented to occupy fewer slots, or that pipeline stalls cause by dependencies between instructions are reduced, or a combination of both.

- **Memory Bottleneck**
  A 3-stage ARM core accesses memory on (almost) every clock cycle either to fetch an instruction or to transfer data. To get significant better CPI the memory system must deliver more than one value in each clock cycle either by delivering more than 32 bits per cycle from a single memory or by having separate memories for instruction and data accesses.

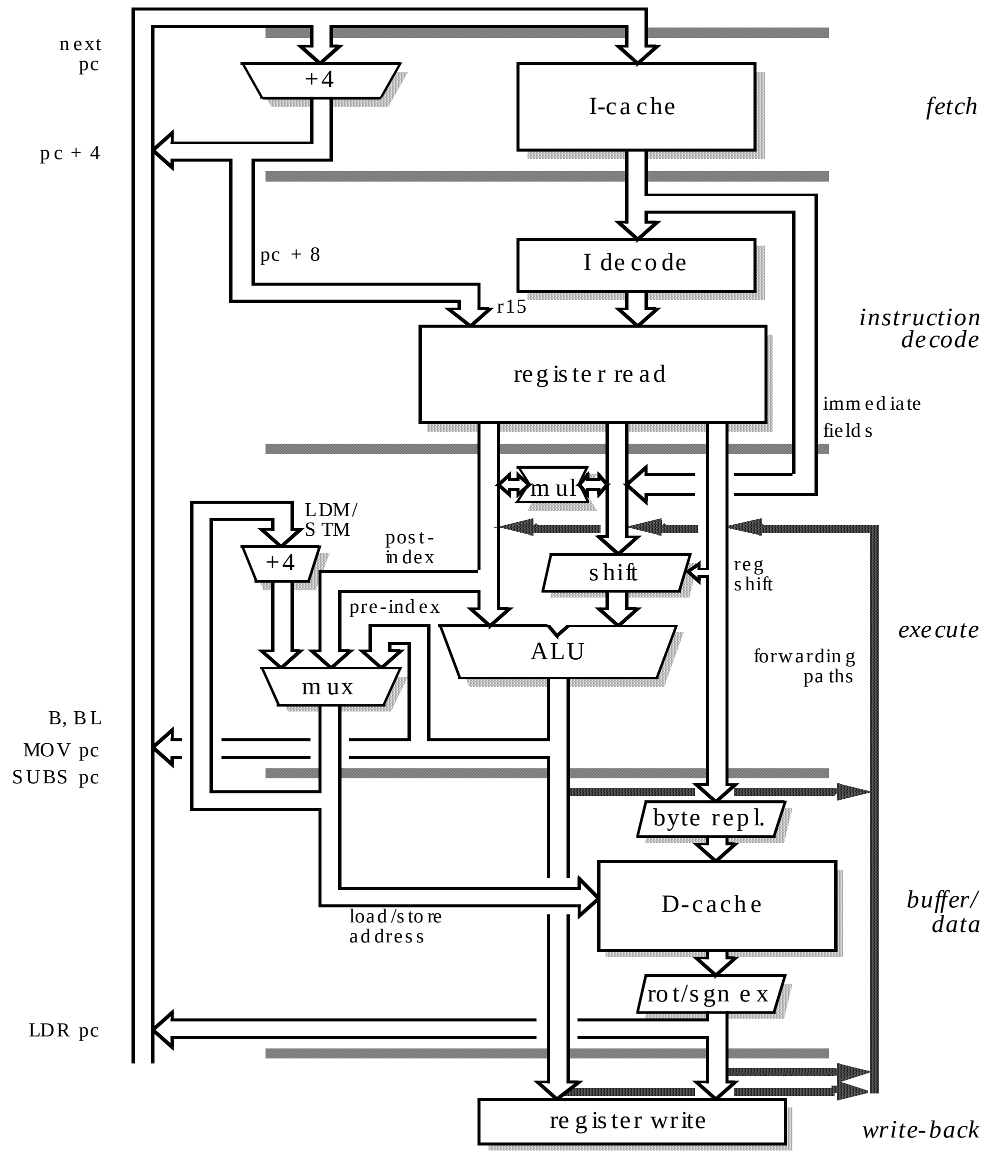
- As a result of the above issues, higher performance ARM cores employ a 5-stage pipeline and have separate instruction and data memories.

The 5-stage Pipeline

- The ARM processors which use a 5-stage pipeline have the following pipeline stages:
  - Fetch
    - The instruction is fetched from memory and placed in the instruction pipeline
  - Decode
    - The instruction is decoded and register operands read from the register file.
  - Execute
    - An operand is shifted and the ALU result generated. If the instruction is a load or store the memory address is computed in the ALU
  - Buffer/data
    - Data memory is accessed if required. Otherwise the ALU result is simply buffered for one clock.
  - Write-back
    - The results generated by the instruction are written back to the register file, including any data loaded from memory

ARM9TDMI 5-stage pipeline organization

- The principal concessions are
- (1) the three source operand read ports and two write ports in the register file (where a ‘classic’ RISC has two read ports and one write port),and
- (2) the inclusion of address incrementing hardware in the execute stage to support load and store multiple data.



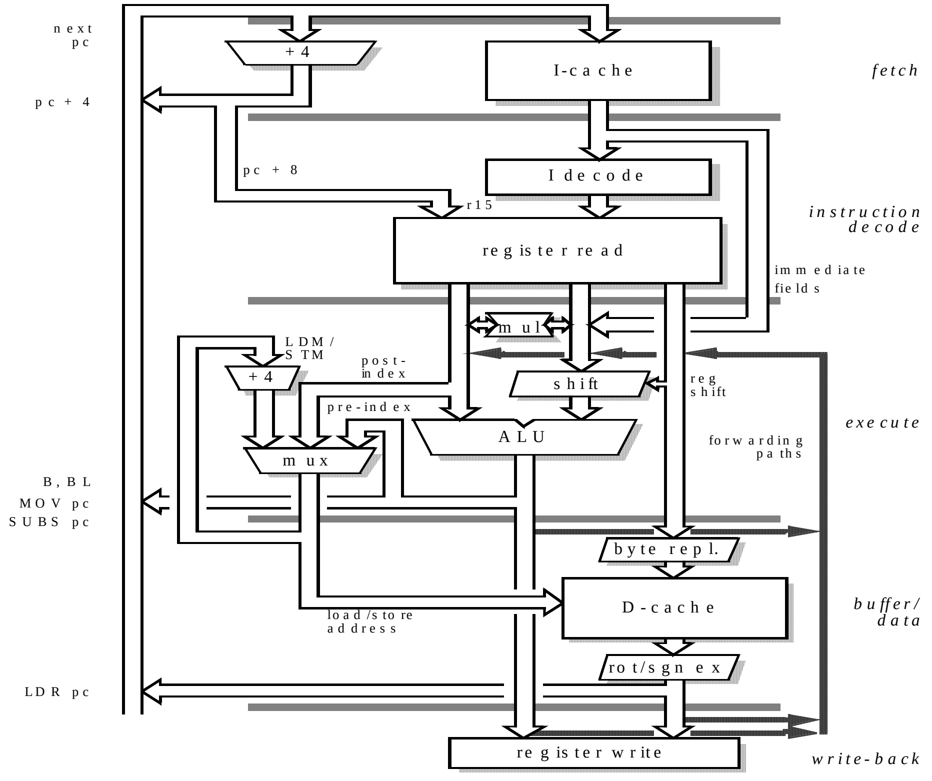
Data Forwarding

- Because instruction execution is spread across three pipeline stages, the only way to resolve data dependency without stalling the pipeline is to introduce forwarding paths. Forwarding paths allow results to be passed between stages as soon as they are available, and the 5-stage ARM requires each of three source operands to be forwarded from any of three intermediate result registers. *(See Fig. 4.4)*

- There is one case where, even with forwarding, it is not possible to avoid a pipeline stall. Consider the following code sequence:

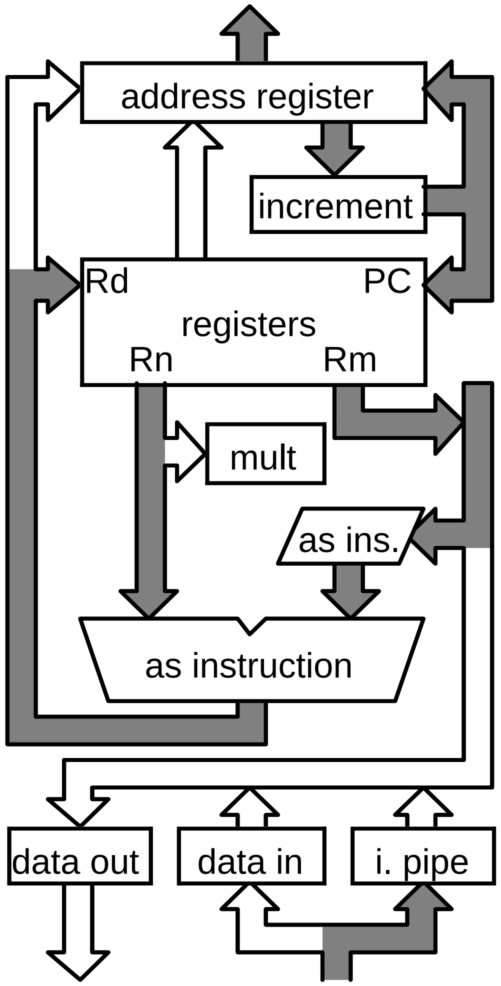
```
LDR     rN, [..]
ADD     r2, r1, rN
```

ADD r3, r1, r2	IF	ID	EXE	MEM	WB				
SUB r5, r4, r3		IF	ID	EXE	MEM	WB			
LDR r6, [r4]			IF	ID	EXE	MEM	WB		
ADD r7, r5, r6				IF	ID	EXE	MEM	WB	
SUB r9, r8, r6					IF	ID	EXE	MEM	WB

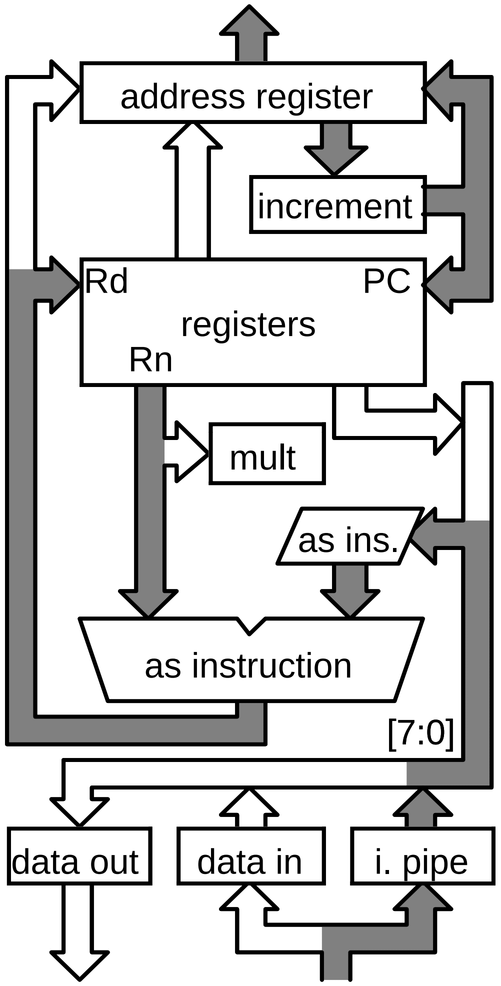


ARM Instruction Execution

- Data Processing Instructions

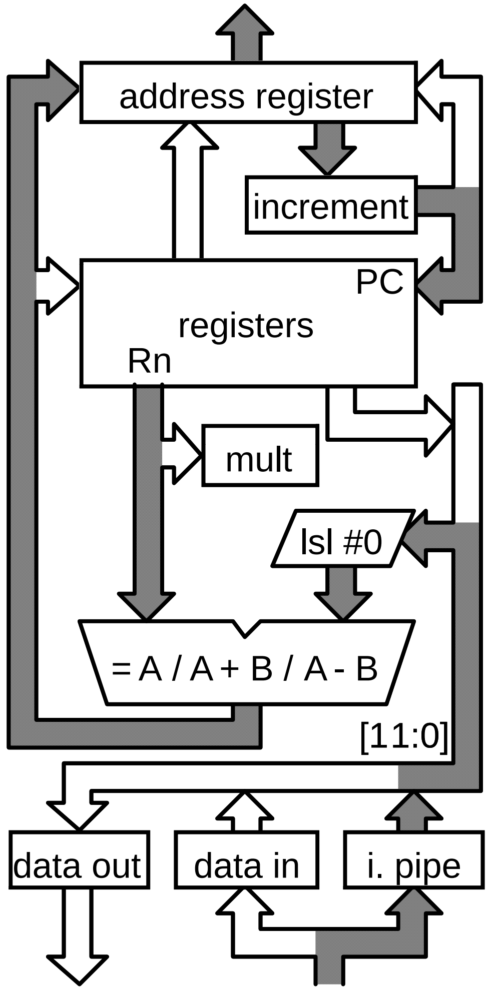


(a) register - register operations

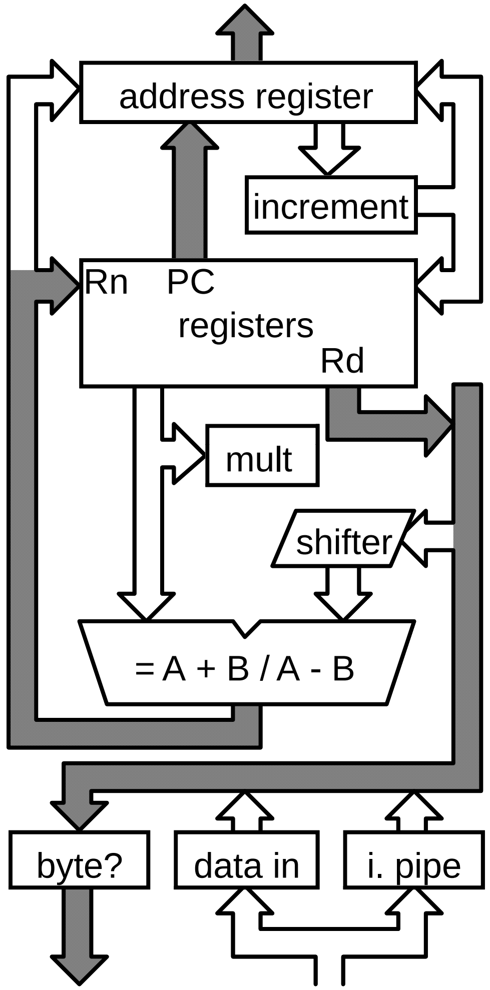


(b) register - immediate operations

❑ Data Transfer Instructions: STR Rd, [Rn, <offset>]!

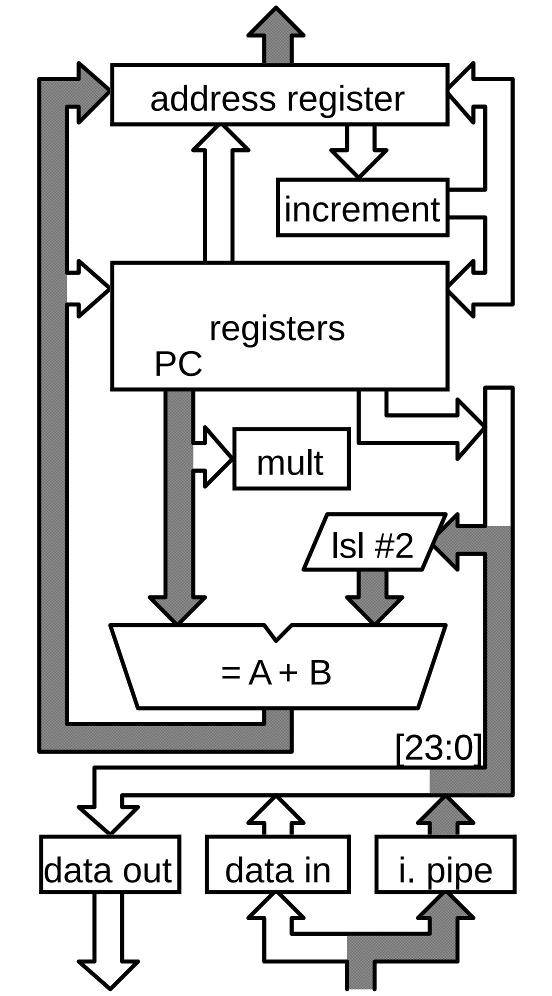


(a) 1st cycle - compute address

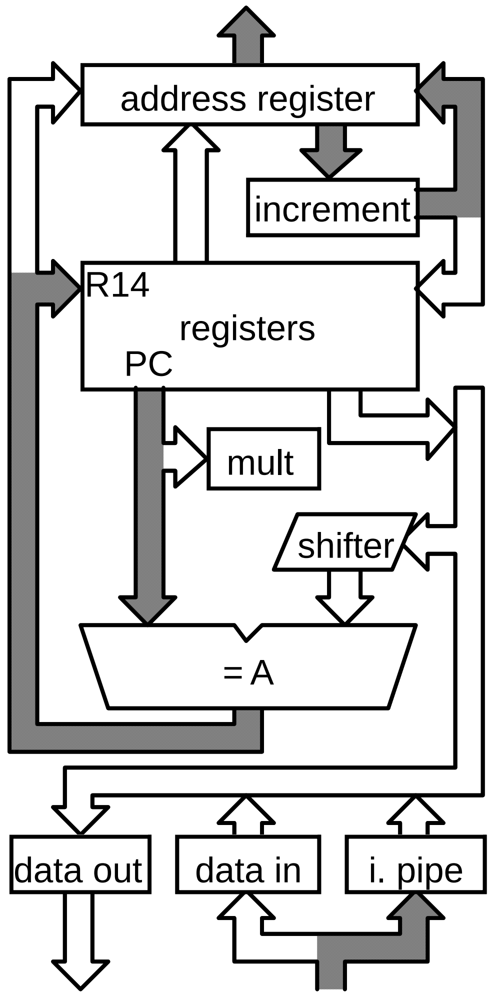


(b) 2nd cycle - store data & auto-index

❑ Branch Instructions



(a) 1st cycle - compute branch target



(b) 2nd cycle - save return address

4.4 ARM Implementation

❑ Clocking Scheme

Unlike the MU0, most ARMs do not operate with edge-sensitive registers; instead the design is based around 2-phase non-overlapping clocks. The non-overlapping property of the phase 1 and phase 2 clock ensures that there are no race conditions in the circuit.

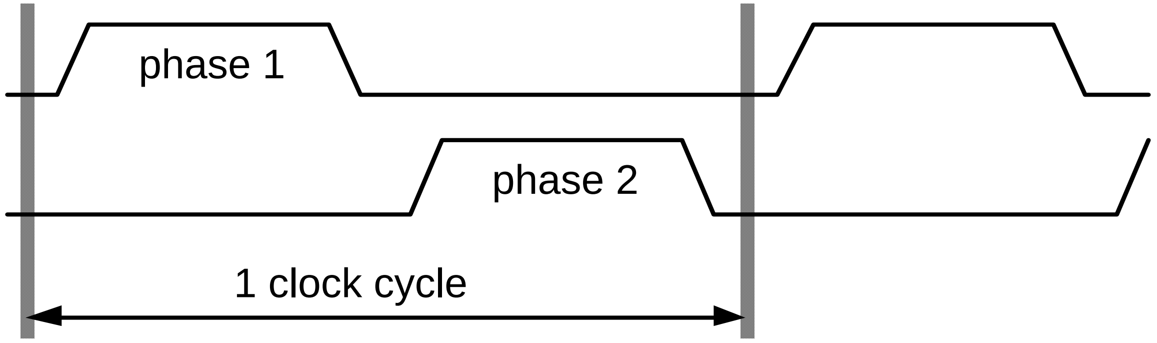


Figure 4.8. 2-phase non-overlapping clock scheme.

4.4 ARM Implementation

❑ Datapath Timing

The minimum datapath cycle time is the sum of :

- The register read time;
- The shifter delay;
- The ALU delay;
- The register write set-up time;
- The phase 2 to phase 1 non-overlapping time.

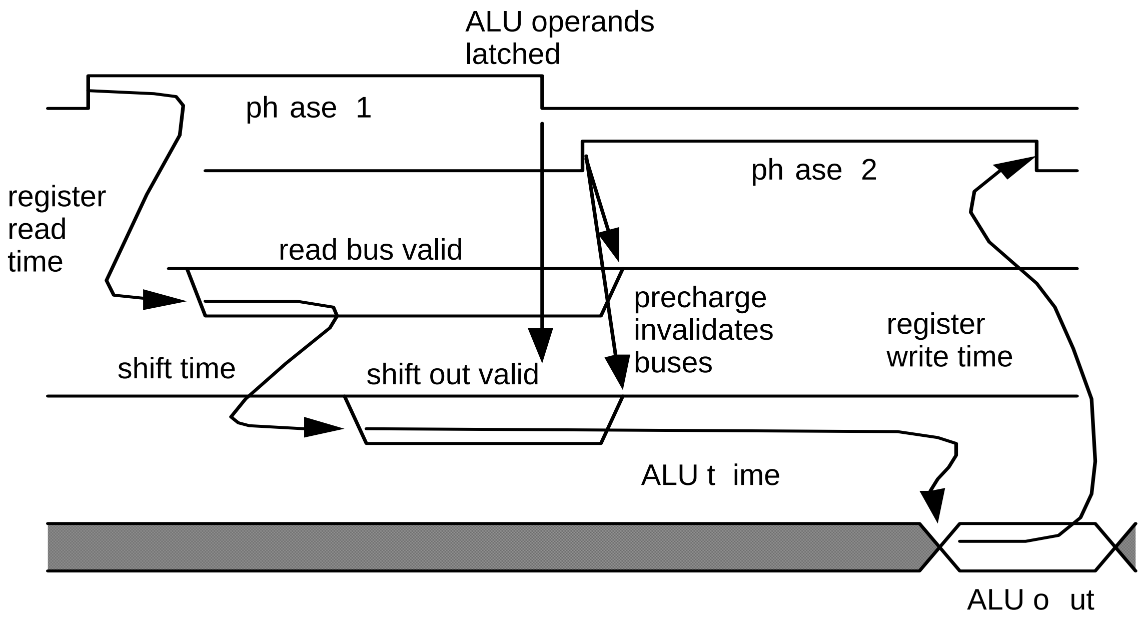
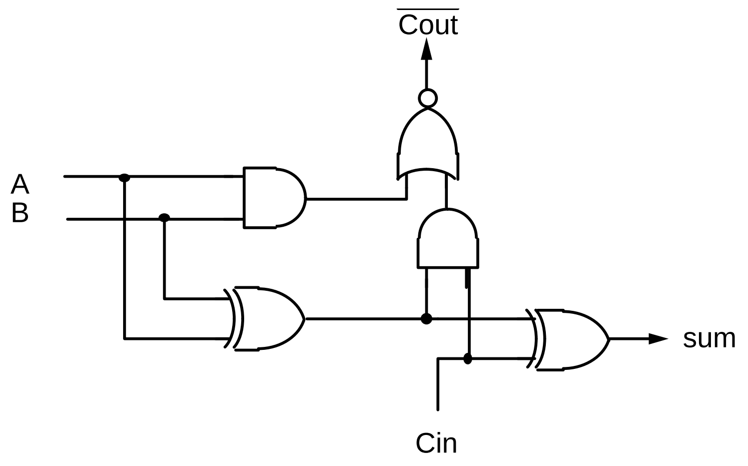


Figure 4.9. ARM datapath timing (3-stage pipeline)

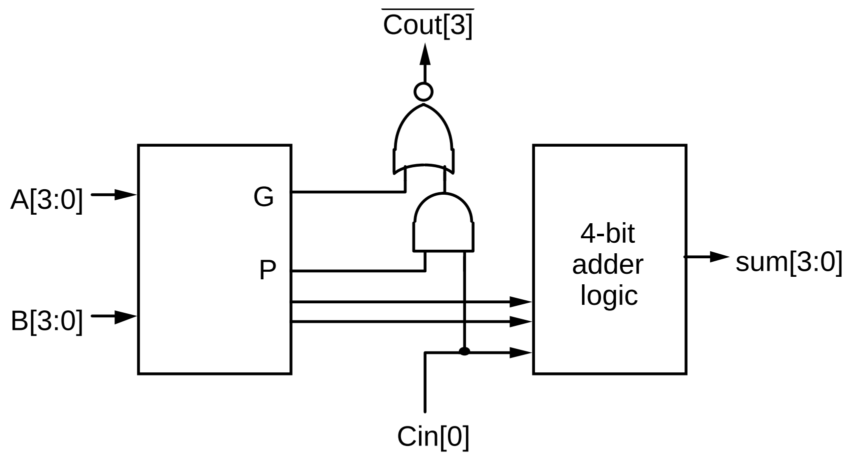
4.4 ARM Implementation

Adder Design

The first ARM processor prototype used a simple ripple-carry adder. The worst-case carry path is 32 gates long.
In order to allow a higher clock rate, ARM2 used a 4-bit carry look-ahead scheme to reduce the worst-case carry path length. The carry propagate path length is reduced to 8 gate delays.



The original ARM1 ripple-carry adder circuit

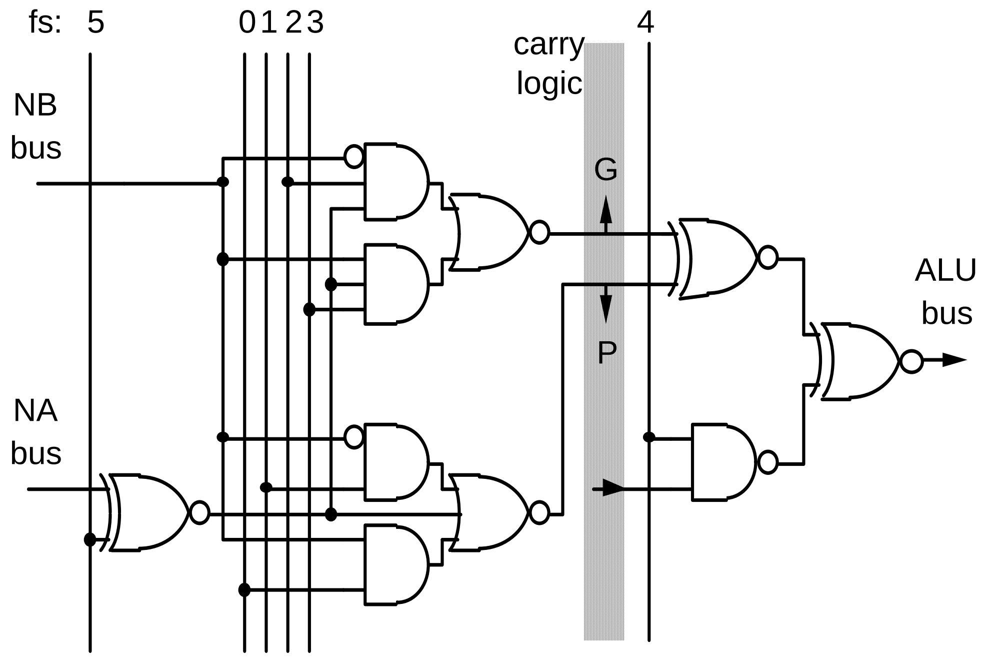


The ARM2 4-bit carry look-ahead scheme

4.4 ARM Implementation

ALU Functions

The ALU does not only add its two inputs. It must perform the full set of data operations defined by the instruction set, including address computations for mem transfers, branch calculations, bit-wise logical functions, and so on.



The ARM2 ALU logic for one result bit

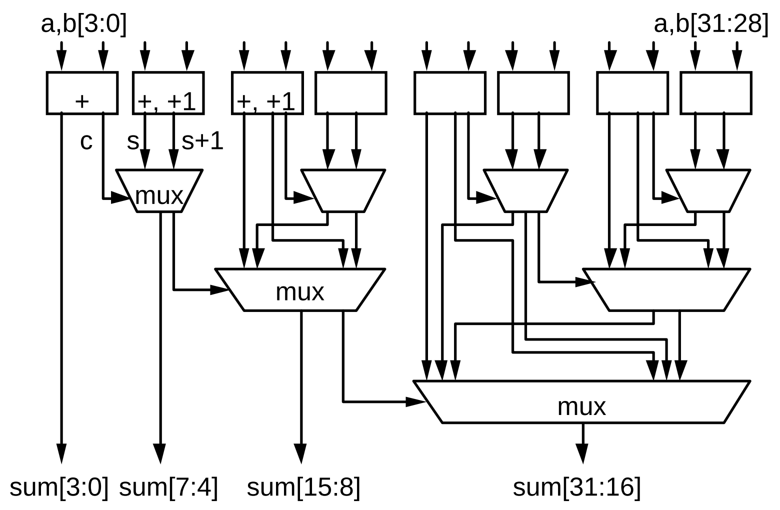
fs5	fs4	fs3	fs2	fs1	fs0	ALU output
0	0	0	1	0	0	A and B
0	0	1	0	0	0	A and not B
0	0	1	0	0	1	A xor B
0	1	1	0	0	1	A plus not B plus carry
0	1	0	1	1	0	A plus B plus carry
1	1	0	1	1	0	not A plus B plus carry
0	0	0	0	0	0	A
0	0	0	0	0	1	A or B
0	0	0	1	0	1	B
0	0	1	0	1	0	not B
0	0	1	1	0	0	zero

ARM2 ALU function codes

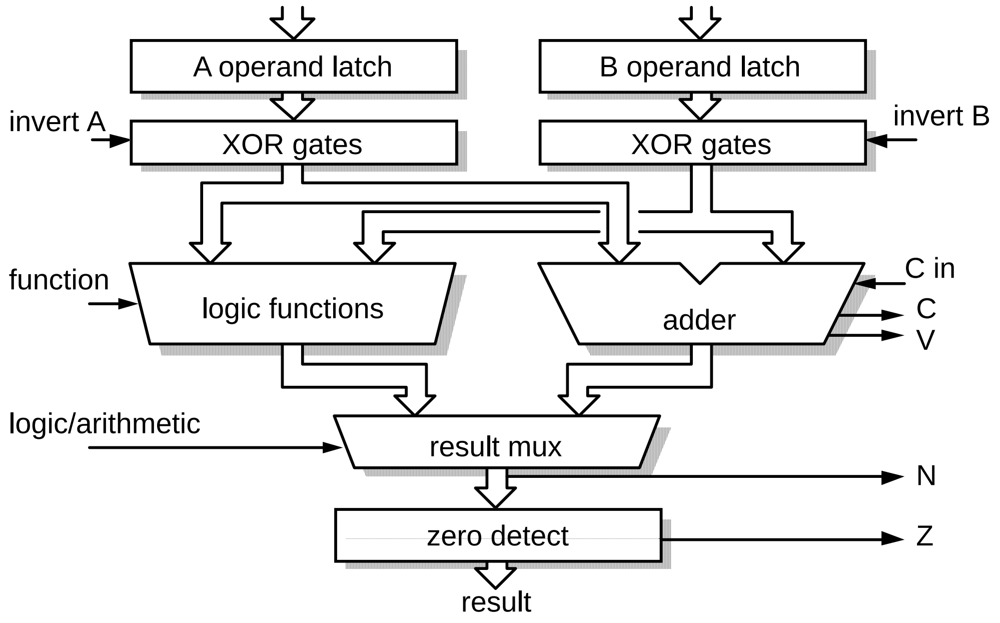
4.4 ARM Implementation

❑ ARM6 ALU Structure

The ARM6 carry-select adder does not easily lead to a merging of the arithmetic and logic functions into a single structure. Instead, a separate logic unit runs in parallel with the adder, and a multiplexer selects the output from the adder of from the logic unit as required.



The ARM6 carry-select adder scheme



The ARM6 ALU organization

4.4 ARM Implementation

- **Carry Arbitration Adder**

  The adder was further improved on the ARM9TDMI, where a ‘carry arbitration’ adder is used. This adder computes all intermediate carry values using a ‘parallel-prefix’ tree, which is a very fast parallel logic structure.

- The carry arbitration scheme recodes the conventional propagate-generate information in terms of u and v. The information can be combined with that from a neighboring bit position using the formula:

$$(u,v)\cdot(u',v') = (v+u\cdot u', v+u\cdot v')$$

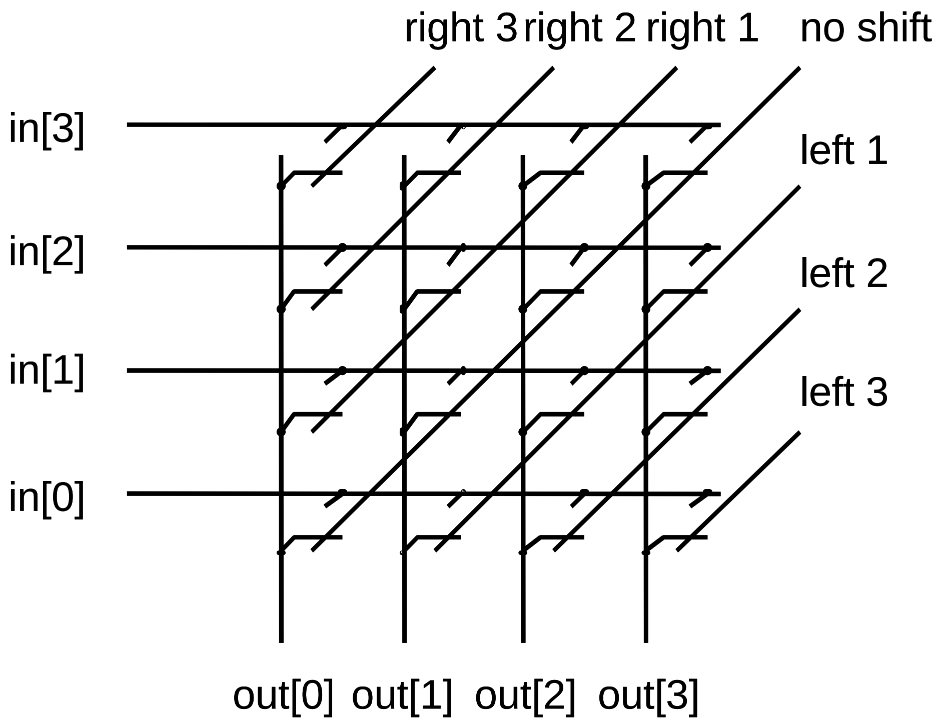
A	B	C	u	v
0	0	0	0	0
0	1	unknown	1	0
1	0	unknown	1	0
1	1	1	1	1

ARM9 carry arbitration encoding

4.4 ARM Implementation

- **Barrel Shifter**

  The shifter performance is critical since the shift time contributes directly to the datapath cycle time. In order to minimize the delay through the shifter, a cross-bar switch matrix is used to steer each input to the appropriate output. (The ARM processors use a 32x32 matrix.) For a rotate right function, …



The cross-bar switch barrel shifter principle

4.4 ARM Implementation

Multiplier design: Low-cost

The older ARM cores include low-cost multiplication hardware that supports only the 32-bit result multiply and multiply-accumulate instructions.

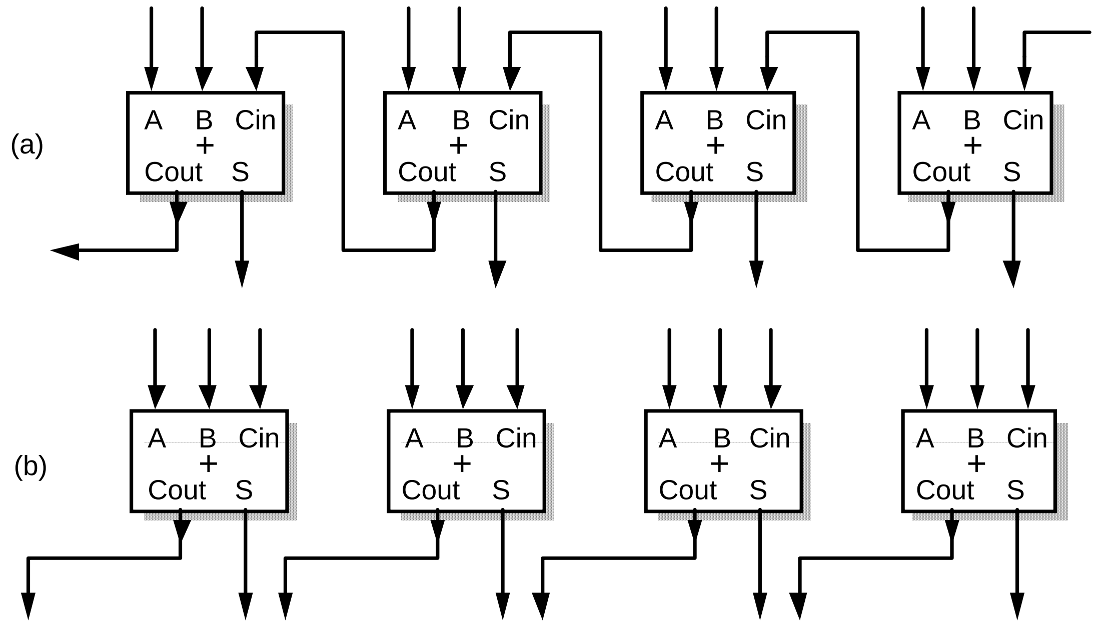
- The multiplier employs a modified Booth's algorithm to produce the 2-bit product, exploiting the fact ×3 can be implemented as ×(-1)+ ×4 . This allows all four values of the 2-bit multiplier to be implemented by a simple shift and add or subtract, possibly carrying the ×4 over to the next cycle.

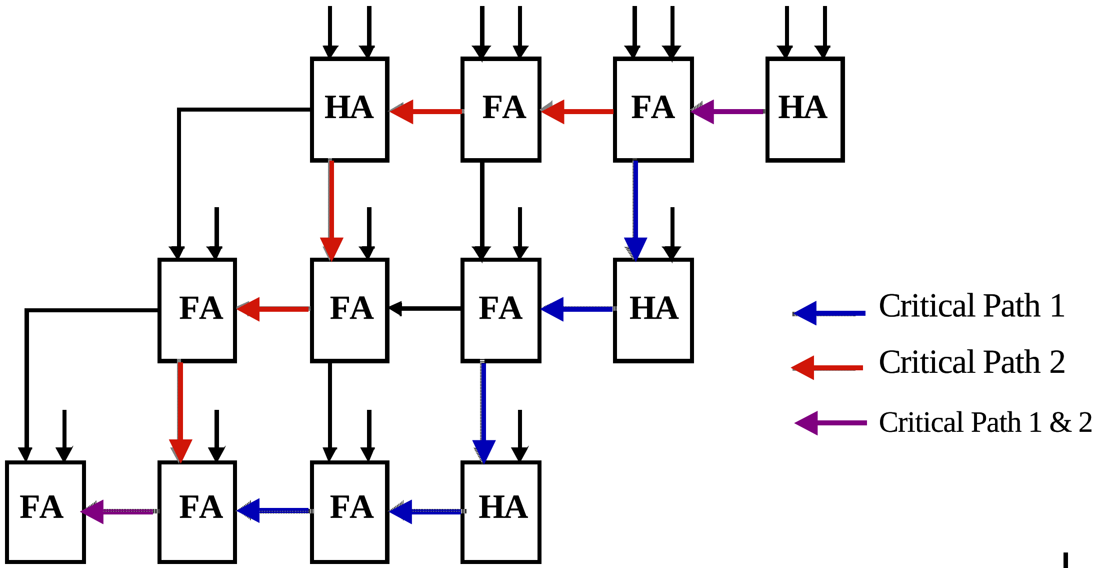
Carry-in	Multiplier	Shift	ALU	Carry-out
0	x 0	LSL #2N	A + 0	0
	x 1	LSL #2N	A + B	0
	x 2	LSL #(2N + 1)	A – B	1
	x 3	LSL #2N	A – B	1
1	x 0	LSL #2N	A + B	0
	x 1	LSL #(2N + 1)	A + B	0
	x 2	LSL #2N	A – B	1
	x 3	LSL #2N	A + 0	1

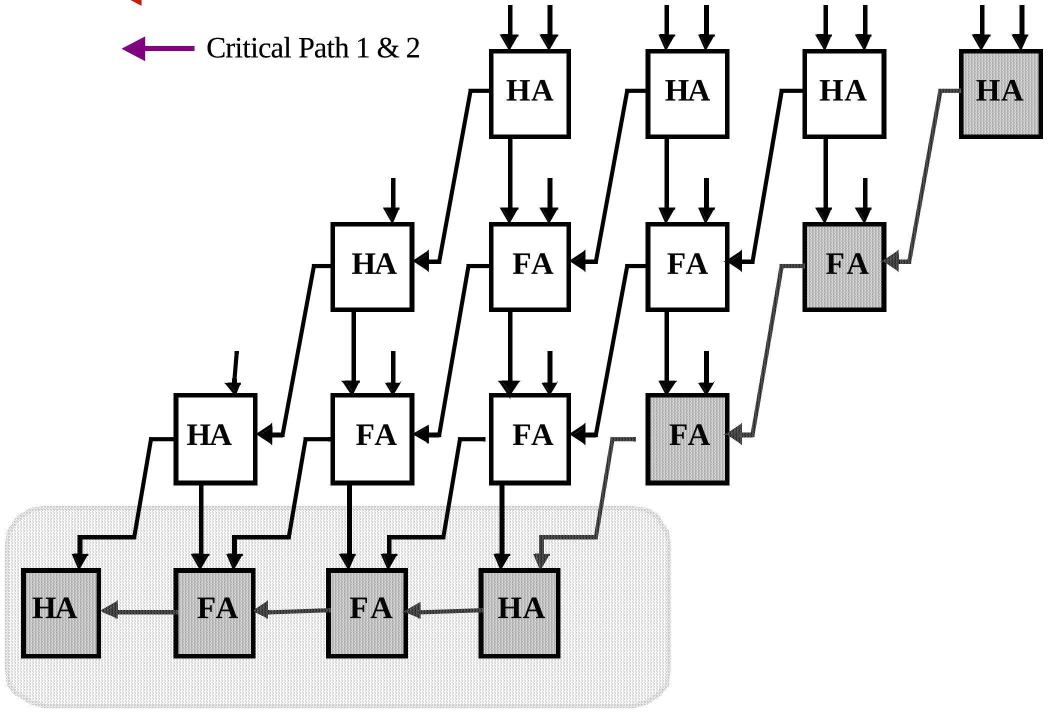
❑ Multiplier design: High-Speed

The high-performance multiplication used in some ARM cores employs a widely used redundant binary representation to avoid the carry-propagation delays associated with adding partial products together.

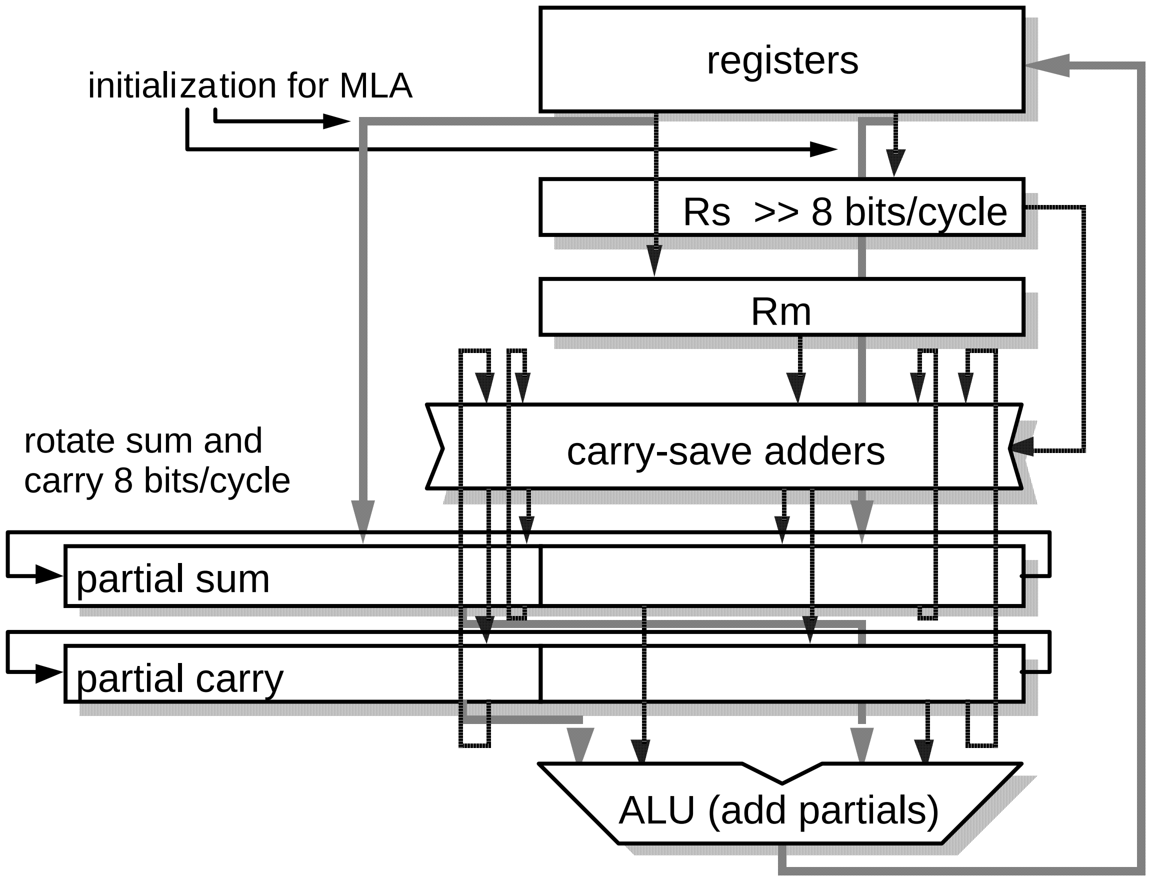
- Intermediate results are held as partial sums and partial carries where the true binary result is obtained by adding these two together, but this is only done once at the end of the multiplication.







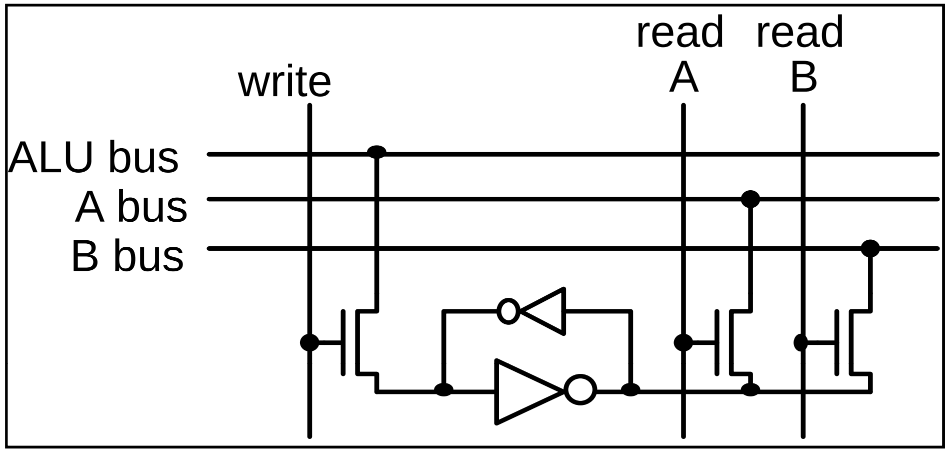
Vector Merging Adder



ARM high-speed multiplier organization

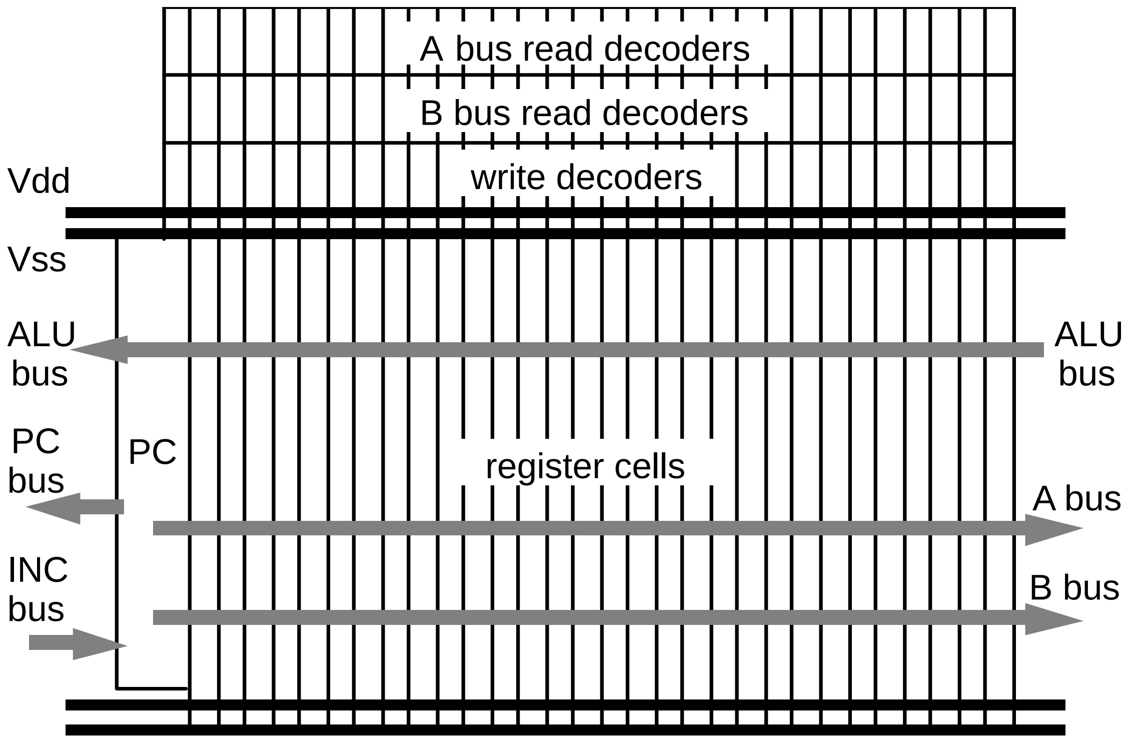
❑ The Register Bank

- The last major block on the ARM datapath is the register bank.
- The storage cell is an asymmetric cross-coupled pair of CMOS inverters which is overdriven by a strong signal from the ALU bus when the register contents are changed. The feedback inverter is made weak in order to minimize the cell's resistance to the new value.
- This register cell design works well with a 5V supply, but writing a '1' through the NMOS becomes difficult at lower supply voltages. ARM cores since the ARM6 …



ARM6 register cell circuit

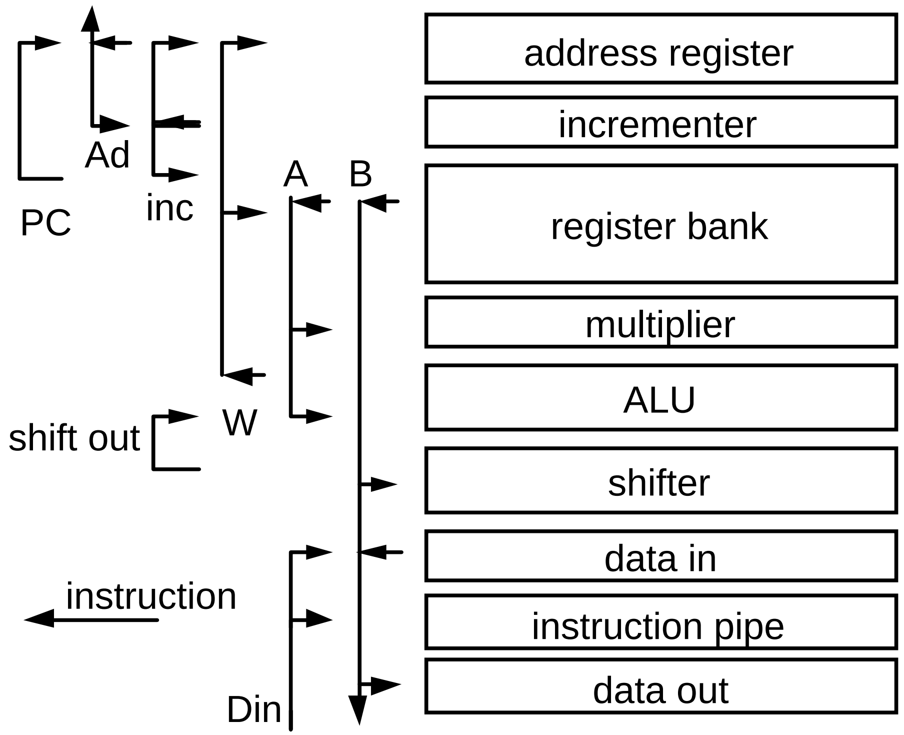
- The ARM PC register has two write and three read ports whereas the other registers have one write and two read ports.
- The symmetry of the register array is preserved by putting the PC at one end where it is accessible to the additional ports and it can be allowed to have a ‘fatter’ profile.
- The register back accounts for around one-third of the total transistor count of the simpler ARM cores, but takes much smaller silicon area.



4.4 ARM Implementation

❑ Datapath Layout

- The order of the function blocks is chosen to minimize the number of additional buses passing over the more complex functions.



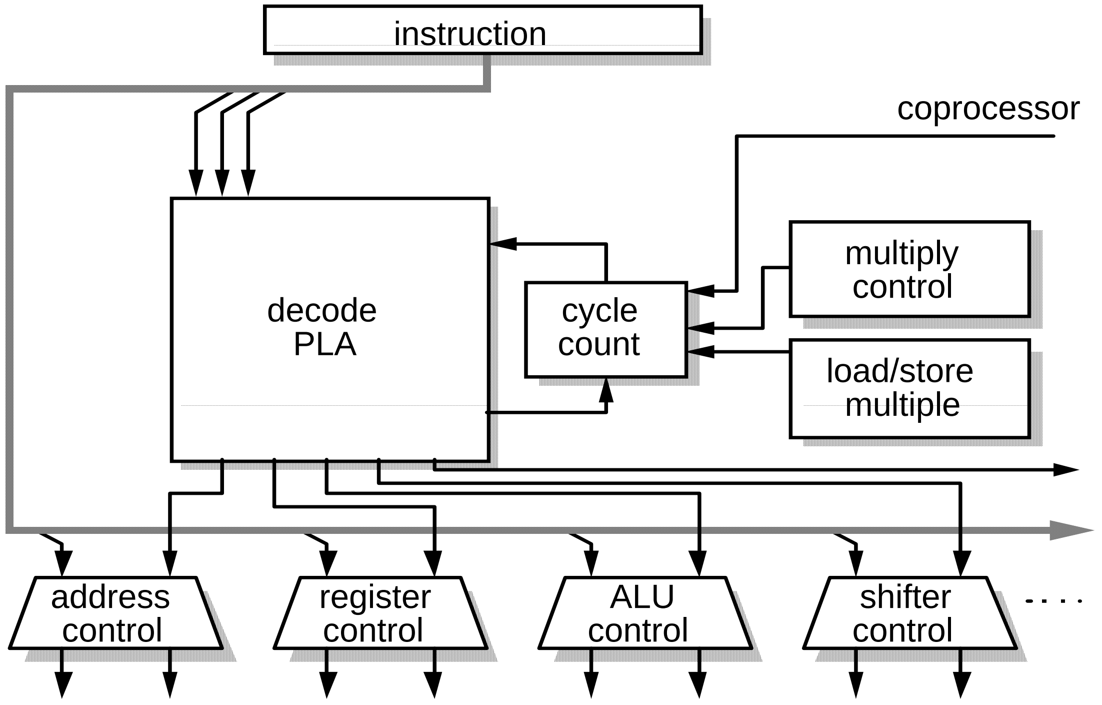
ARM core datapath buses

4.4 ARM Implementation

❑ Control Structures

The control logic on the simpler ARM cores has three structural components which relate to each other.

- An instruction decoder PLA.
- Distributed secondary control associated with each of the major datapath function blocks.
- Decentralized control units for specific instruction that take a variable number of cycles to complete.



4.4 ARM Implementation

- **Physical Design**

  There are two principal mechanisms used to implement an ARM processor core on a particular process;

  - A hard macrocell is delivered as physical layout ready to be incorporated into the final design.
  - A soft macrocell is delivered as a synthesizable design expressed in a hardware description language such as VHDL.

4.5 The ARM Coprocessor Interface

- The ARM supports a general-purpose extension of its instruction set through the addition of hardware coprocessors.

- **Coprocessor Architecture**

  The most important features of coprocessor architecture are:
  - Support for up to 16 logical coprocessors.
  - Each coprocessor can have up to 16 private registers of any reasonable size; they are not limited to 32 bits.
  - Coprocessors use a load-store architecture, with instructions to perform internal operations on registers, instructions to load and save registers from and to memory, and instructions to move data to or from an ARM register.

ARM7TDMI Coprocessor Interface

- The ARM7TDMI (Thumb, Debugger, Multiplier, ICE) coprocessor interface is based on ***'bus watching'***. The coprocessor is attached to a bus where the ARM instruction stream flows into the ARM, and the coprocessor copies the instructions into an internal pipeline that mimics the behavior of the ARM instruction pipeline.

- As each coprocessor instruction begins execution there is a ***'hand-shake'*** between the ARM and the coprocessor to confirm that they are both ready to execute it. The handshake uses three signals:
  - *cpi* ('CoProcessor Instruction', from ARM to all coprocessors).
    This signal indicates that the ARM has identified a coprocessor instruction and wishes to execute it.
  - *cpa* ('CoProcessor Absent', from the coprocessors to ARM).
    This signal tells the ARM that there is no coprocessor present that is able to execute the current instruction.
  - *cpb* ('CoProcessor Busy', from the coprocessors to ARM).
    This signal tells ARM that the coprocessor cannot begin executing the instruction yet.