One ISA, many implementations

Technology scaling: happening anyway … Moore's Law

Once upon a time as technology allowed more sophisticated processors …

… a new architecture was defined every few years by each competing manufacturer.

The software had to be rewritten … but there wasn't that much and it was time to upgrade, anyway.

Then people wanted to keep some of their familiar software – and they'd paid for it.

So the ISA had to stay the same … or, at least, be backwards compatible.

So the engineers have to spend their transistors on making machines that run old instruction sets … but faster.

- Higher clock rates: part technology, part **pipelining**
- **Parallelism**: more functional units; more cores

This lecture uses (a subset of) the ARM development to illustrate features and principles.

Why does this matter?

Historically: Evolution

- More transistors available: an opportunity for speed up
- More designer experience/tools: can do more in development time
- More instructions: new applications drive customer 'wants'
- Backward compatibility: the desire (economic *need*!) to keep supporting an earlier instruction set – even an earlier architecture
  - e.g. x86 support for 32- and 16-bit ISAs on 64-bit architecture

Contemporarily: Trade-offs

Demands are typically for high performance, low energy and small silicon area (because area equates to production cost). Usually small implies low energy but also slow; spending on extra silicon allows a performance increase but tends to increase energy use too.

This is not a hard-and-fast rule, however. For example, enlarging a cache usually increases its hit rate (up to a point) which will increase performance *and* save energy by reducing the number of accesses to the (more power-hungry) main memory.

The architecture designer has to balance these variables to find a good solution for the intended application(s).

This can result in a range of code-compatible devices with different microarchitectures.

ARM's big.LITTLE™

big.LITTLE™ is ARM's name for an energy-saving scheme which employs two code-compatible ARM cores. One of these is a high-performance system, the other is a small, low-power core which omits speculation etc. Together these run a single process (at any instant) and are scheduled by system software so that the 'little' core runs when there is little or no processing to do but hands over to the 'big' core when it can no longer cope with the processing demand. This is useful in a real-time system as it can provide high performance when needed but revert to low-power operation when the demand is lighter; the object code is the same in both cases.

How to read this set of notes

This lecture uses ARM examples of stages of evolution of a processor. These are *illustrations* of how things have been introduced; specific features need not be memorised. The important issues are things like the trade-off of hardware cost vs. clock speed. vs. cycle count vs. power consumption.

Power and energy

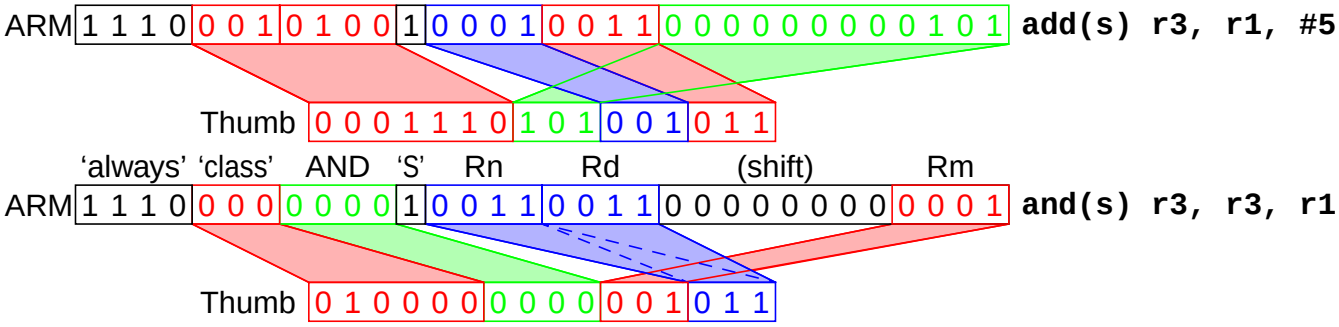
Power is energy/time. Power matters because the energy has to get to the circuit (as electric current, requiring wires) and then dissipate as heat; the heat dissipation is the more significant problem in most cases. Power requirements can be reduced by computing more slowly.

Energy matters in battery-powered equipment because there is only so much energy a battery can hold. Slowing down simply uses the same energy over a longer time for the same computation.

However, the *mechanisms* used to obtain very high performance typically involves extra circuits which use more energy overall, so fast designs typically use more energy than slow ones.

Thumb

'Thumb' is ARM's version of an alternate instruction set *coding*. Examples:



Originally a set of 16-bit codes which specified the 'most useful' subset of ARM operations.

Thumb runs on the **same underlying architecture** as ARM: i.e. 16 32-bit registers (with R15 as PC, R14 as LR, R13 as SP), load-store operations etc.

- Thumb was originally motivated to increase **code density**
  - i.e. get code with the same functionality into a smaller space.
- Smaller memory ⇒ smaller chip ⇒ lower cost
- This can also allow more efficient use of off-chip memory (ROM) with a 16-bit interface.
- More instructions ⇒ longer execution time

Thumb

Roughly speaking Thumb imposes the following restrictions:

- Only R0-R7 can be used (for most operations)
- Most data operations are two operand
  - i.e. one of the sources must be the destination
- Shifts and rotates are separate instructions
  - no space in op. code to specify both together
- Immediate (literal) values are shorter
  - Most branches are short
- Most operations are not conditional
- Flag setting is not optional
- Some operations are omitted

This reflects the characteristics of 'typical' code

- A source register is also the destination ~50% of the time
- Most immediate values are small (typically '0' or '1'; a few others)
- Most branches are short: think of loops and 'if's
- Some data operations are very common {MOV, ADD, SUB, CMP}, others are rarely used

Outcome

Thumb code needs more instructions to achieve the same end: about 40% more.

Thumb instructions are half the size of ARM instructions.

Time taken ≈ 100% + 40% = 140% Code size ≈ 140%/2 = 70%

So: Thumb code is **slower** but **smaller**.

It is possible – and, sometimes, desirable – to **interwork** Thumb and ARM code. This can be done at the procedure level with dedicated 'call' ('BLX') instructions. When this is done it is normal to generate an execution profile of the code, typically by simulation. time-critical or frequently used code can use the ARM instructions, infrequently used code (e.g. error handling) can be in Thumb to save space.

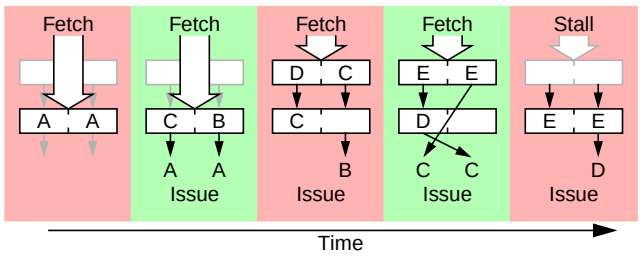
Thumb2

As introduced, Thumb could not do all the operations needed for a 'complete' processor. For example it does not contain the (rarely used) multiprocessor locking operations. For certain facilities it was necessary to switch back into the full ARM coding. This imposes a performance (and organisational) penalty in dividing code into sections using different binaries and switching between them.

Thumb2 addressed this by extending the Thumb codes to cover other operations. To do this requires some longer instructions, so Thumb2 (which is the current 'Thumb' coding) is a mixture of 16-bit instructions from the original Thumb and newly recoded 32-bit instructions.

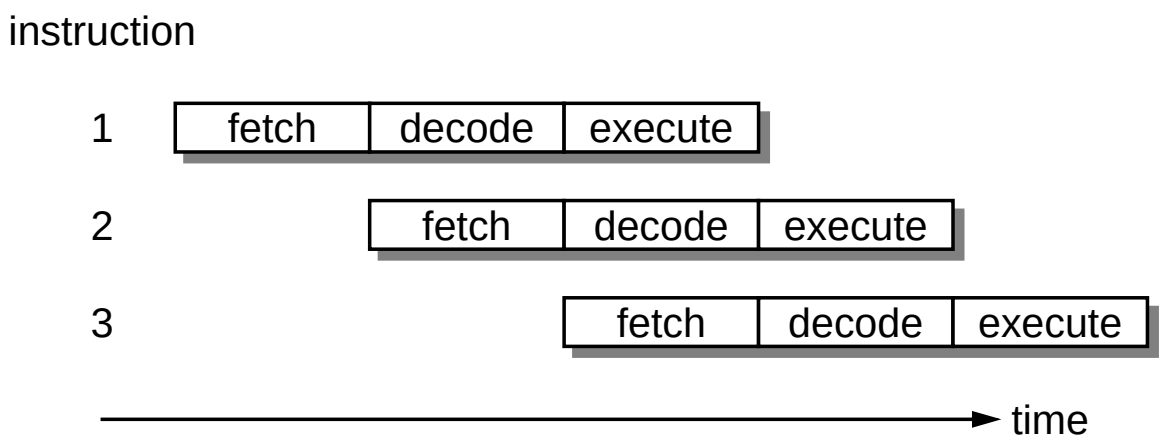
To make this fit, the 32-bit codes *do not have to be word aligned*, so may be fetched in two separate memory cycles.

This, of course, increases the complexity of the processor. To keep things efficient an implementation will fetch a 32-bit word and then see if it is a 16- or 32-bit instruction. If it is the latter it can issue it whilst fetching the next word; if it is the former it can issue the relevant half while fetching the next word. If the second instruction is also 16-bit the prefetch may then be stalled: if it is 32-bit the next instruction can be assembled straight away.



This is an example of a **prefetch buffer**, where the instruction fetch process is only loosely coupled to the execution in a pipeline. The operation should be deductible: words are fetched from memory if the buffer is not full; the next instruction is identified, assembled (using multiplexers) and issued if it is complete.

The 3-stage ARM pipeline



- Single cycle instructions
  - complete at a rate of one per clock cycle
  - intended to use (a single) memory efficiently

The 3-stage ARM pipeline

(The original architecture which has affected on the instruction set.)

- **Fetch**
  - the instruction is fetched from memory
  - timing: involves the 'long' wait for the memory
- **Decode**
  - the instruction is decoded and the datapath control signals prepared for the next cycle
  - timing: quite short. ARM instruction decoding is reasonably simple (or was before the instruction set was extended)
- **Execute**
  - the operands are read from the register bank, shifted, combined in the ALU and the result written back
  - timing: includes register reading, the shifter, the ALU and the register writeback delays *in series*; a comparatively long cycle time

This results in a somewhat unbalanced pipeline as the decoder stage should have some 'slack' time. However it is still faster than including the decoding in the (already slow) 'execute' stage.

The execution stage is slow but it results in a fairly simple design: all the operations execute in one cycle which means the operands for the following instruction will all be available in the next cycle.

Consider:

```
ADD   R0, R0, #1
CMP   R0, #10
```

The ADD reads a value from R0 and then overwrites it with a new value.

The CMP must read the new value; in the model above this is guaranteed to be in the register bank when it arrives to execute.

The 3-stage ARM pipeline

- PC behaviour
  - r15 increments twice before an instruction executes
    - due to pipeline operation
  - therefore r15 = address of instruction + 8
  - normally the assembler makes the necessary adjustments, e.g. in branches

This *behaviour* is **consistent for all ARMs**, although the pipeline structures may vary.

Later microarchitectures have to 'pretend' to have the same pipeline to ensure that the binary object code will behave in the same way.

MIPS

MIPS – originally 'Microprocessor without Interlocked Pipeline Stages' – is a classic RISC design which exposed the microarchitecture at ISA level. The philosophy pipeline more deeply for speed, which could create inter-instruction dependencies, then resolve potential problems by reordering the object code so that a register read could only take place after it was certain that the value was up to date. (In extremis, NOPs could be inserted.) This avoided solving dependencies dynamically, in hardware.

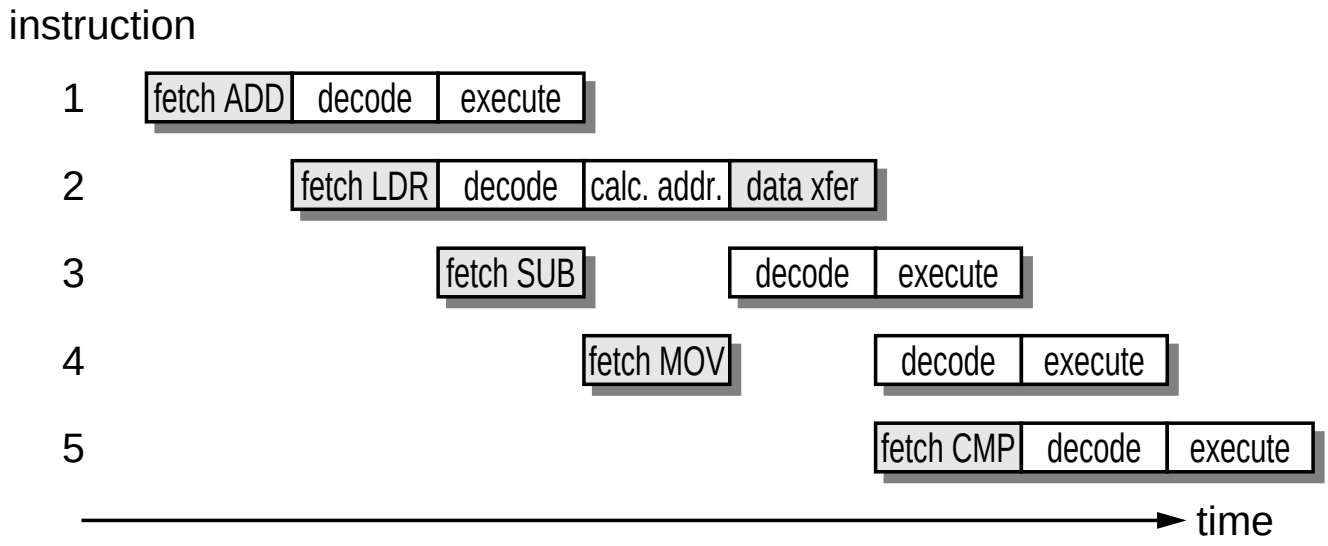
A flaw in this approach is that a different pipeline structure may change the number of cycles to 'retire' (complete) an instruction, thus requiring a recompilation.

Another interesting feature of MIPS is the **deferred branch**. To alleviate the cost of pipeline flushes following a branch, the branch can be moved up in the object code and the following instruction executed *even if the branch is taken*. This location is known as a 'delay slot'.

```
j     address          ; Jump (unconditional)
ADD   $1, $2, $3       ; Executed anyway
```

The 3-stage ARM pipeline

- More complex instructions:



- 'LDR' causes a stall while transfer occurs
- Multi-cycle operations (e.g. multiply) also cause a stall
- 'LDM'/'STM' stall for a number of cycles

The 3-stage ARM pipeline

This model was designed before on-chip caches were common and was intended to make best use of a single memory bus. When a load or store was required this, necessarily, prevented a prefetch operation.

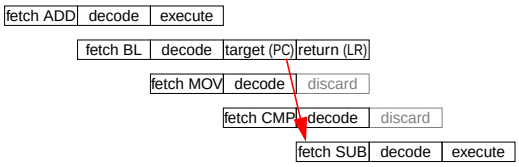
Memory operations are shaded on the slide. Note that the memory stays busy in every cycle.

The memory operation means that load or store take an extra cycle to complete; note that it is complete before the next instruction starts, so that can use any loaded data painlessly.

LDM/STM require multiple, successive data transfer cycles; these can be pictured by extending the stalled period in the picture.

Branch and Link (BL)

Execution of BL is more complicated because, with a single ALU, it requires two cycles to execute. The first cycle uses the ALU to calculate the new PC by adding the offset to the PC value as read. This is done first so the prefetch can start refilling the pipeline.



In the example the MOV and CMP follow the BL and are speculative: they can be discarded by the execute stage which can, instead calculate the return address. A calculation is necessary because the PC in the prefetch unit will have been incremented further so its value needs decrementing again to find the address following the BL (the address of the MOV, in this example).

[Don't worry too much if that is hard to follow; it's the *principle* that operations need identifying and mapping into cycles that matters.]

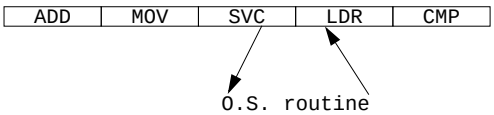
Bonus information

[*This page was looking blank so here is some rather excessive detail. Don't feel you have to memorise all this. It's about ARM's exception mechanisms.*]

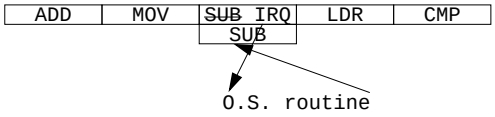
Exceptions

Hardware exceptions are things like interrupts. They cause lots of problems for the microarchitect because they disrupt the normal flow of instructions down the pipeline. Here are a couple of examples:

SVC (a.k.a. 'SWI') is a supervisor call: it is not really an "exception" as it's an instruction in the code but is handled in the same way as other exceptions. It causes a 'call' to a routine – similar to the more familiar BL – but the routine must be part of the O.S.



Interrupt can enter at the decoder by 'hijacking' an instruction which has already been fetched, causing it to decode to similar operations to SVC. The major difference is that an instruction was overwritten so the routine has to return to an earlier address. This is handled in software.

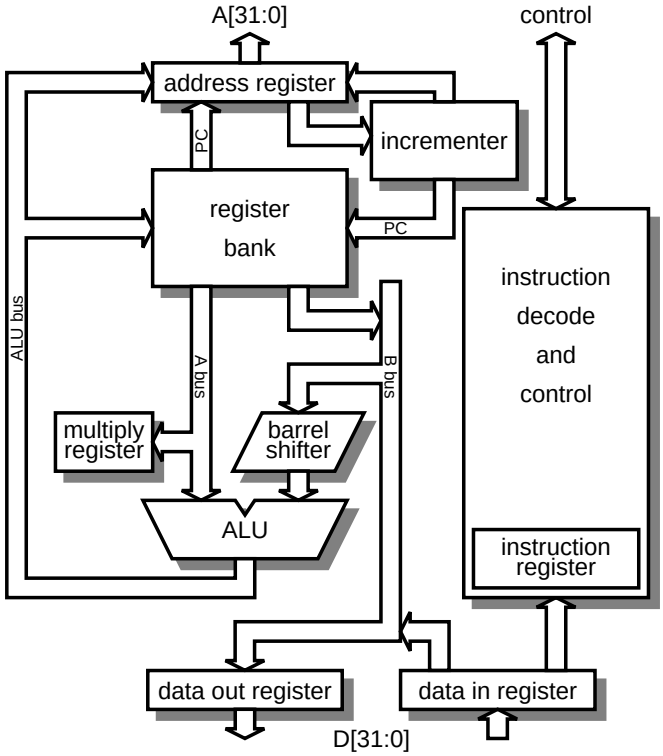


Data abort occurs when a page fault (or similar) prevents a memory transfer from completing, i.e. at the end of the data transfer cycle. It is handled like the other exceptions above (keeping the hardware simple/cheap). However another fetch (#4 on the slide) will have taken place so the 'return address' – used to diagnose which instruction faulted and may be retried – is incremented further than in the other exceptions. This is another case where the (first) microarchitecture has affected the instruction behaviour for ever after.

'Correction' for the offset is again left to software, for economy.

3-stage ARM organization

- Separate address incrementer
- Two register read ports
- Barrel shifter in series with ALU
- Multiplier
  - two bits at a time (multicycle)
  - early termination (unsigned)



The 3-stage ARM organization

Used for all the early ARMs. There was no ARM4[1] or ARM5[2].

Part	Architecture	Comments
ARM1	v1	Prototype only
ARM2	v2	First commercial part
ARM3	v2a	ARM2 + cache
ARM6	v3	Widely successful
ARM7	v3-v4	Updated/improved ARM6

This approach has the advantage that *execution* is not pipelined so register hazards are avoided.

Multiplication

Multiplication is repeated addition but it can be accelerated by shifting and adding. For example, to multiply 'x' by 5 (binary 101) one can add 'x' to zero 5 times, or add 'x'-shifted-left-two-places and 'x' to zero. For an N-bit multiplier the first method may require up to 2^N additions whilst the second only needs 'N'.

$x = 13_{10} = 1101_2$

$$\begin{array}{r} 1\ 1\ 0\ 1 \\ +\ 1\ 1\ 0\ 1\ 0\ 0 \\ \hline 1\ 0\ 0\ 0\ 0\ 0\ 1 \end{array}$$

There are also mechanisms for doing more than a single add in a cycle.

The first ARMs did two-bits-at-a-time multiplication using the datapath shifter/adder combination: a general 32-bit multiplication therefore took 16 cycles (actually plus a cycle or two extra). This implies a pipeline stall controlled by a state machine. Early termination is possible by spotting that all the remaining bits to multiply by are zero (thus the answer won't change further) and stopping.

1. With regard to *tetraphobia*; commonest in East Asia
2. Don't know why not.

This means a bit more logic in the state machine; it also means that the multiplier behaviour is asymmetric (in time) so that 12345×5 may be faster than 5×12345.

ARM9 does 8-bits-at-a-time with a consequent speed up in (most) multiplications. ARM11 pipelines its multiplications (using a separate, parallel pipeline) so it can start a multiplication on every instruction.

A floating point unit may (for example) use some combination of iteration and pipelining but, again, benefit from parallelism because not every instruction tries to use it.

Barrel shifter

Many microprocessors only allow shifts operations one-bit-place-at-a-time. If multiple-place shifts are wanted they have to be performed iteratively, requiring a number of cycles and an FSM or a software loop. The hardware is relatively cheap, however.

A 'barrel' shifter is a device capable of shifting any number of places in 'one go', i.e. within a single cycle.

For a 32-bit machine this could be done with thirty two 32-input multiplexers, allowing arbitrary (but interrelated) selection of the source of each output bit. However a 32-input multiplexer is infeasible as a single logic stage in many technologies and a cascade of gates soon gets expensive.

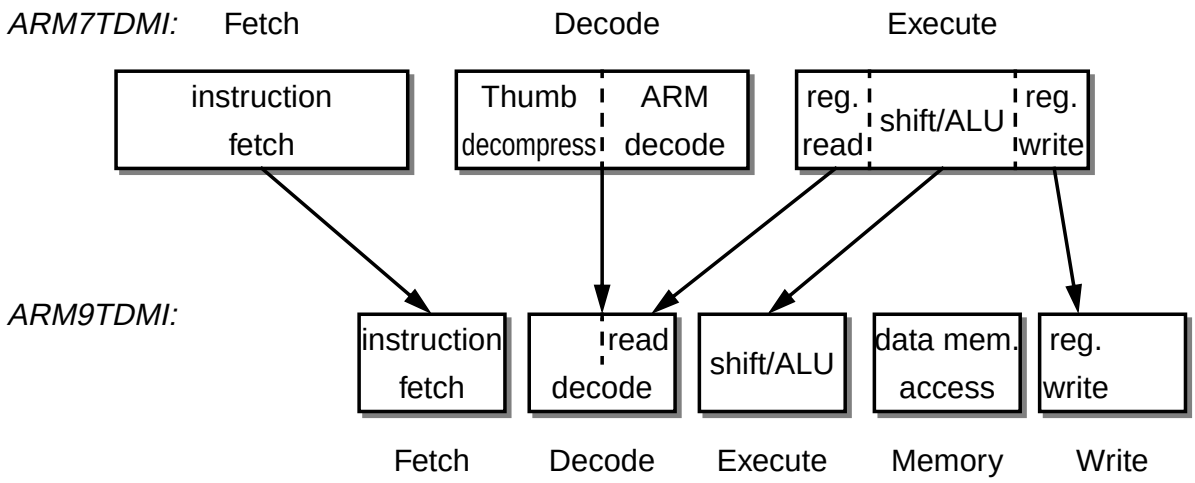
[Made from 2-input multiplexers, a 32-input unit requires 16+8+4+2+1 = 31 units, thus 992 for the whole unit.]

A typical way to implement a barrel shifter is to shift (or not) by powers of two. Thus for a 32-bit shifter there would be five stages, allowing shifts of 16, 8, 4, 2 and one place successively. Programmed with the number of places (in binary) 0..31 shift places (in a given direction) can be achieved with $5 \times 32 = 160$ 2-input multiplexers.

It is possible to make a much more efficient unit by designing at **transistor-level**. Unfortunately this will **not** be very **portable** between fabrication plants, nor onto new processes so the design effort required strongly discourages this for most ASIC applications.

ARM9TDMI pipeline

- The next (popular) ARM device moved to a five stage pipeline



- Thumb instructions are decoded directly
- 'Execute' pipelined ⇒ potential dependency hazards
- 'Harvard' architecture, as far as the caches: parallel instruction & data fetches

The 5-stage ARM pipeline

The ARM9 is a 'classic' five-stage pipeline

- Fetch
  - manage PC; speculatively read *instruction* memory
- Decode
  - instruction decode and register read
  - note any needs to stall or forward and adjust issue accordingly
- Execute
  - commit to competing instructions (unless data fault occurs)
  - shift and ALU; derive addresses
  - multi-cycle for operations such as multiply: stalls pipeline
  - dispatch branches to fetch unit
- Memory
  - *data* memory access; NOP for 'internal' instructions
- Write-back
  - complete to updating registers; processor state changed

Instruction issue is in-order.

Forwarding is used to avoid most stalls.

There is a latency for loads: static code reordering (by compiler) can alleviate this. Example:

```
LDR   R0, [R4]      ; Initiate load
ADD   R0, R0, #1    ; Must stall for load
LDR   R1, [R4, #4]  ; Initiate load
CMP   R1, #0        ; Must stall for load
```

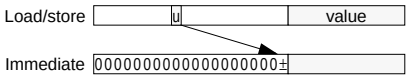
can become:

```
LDR   R0, [R4]      ; Initiate load
LDR   R1, [R4, #4]  ; Initiate load
ADD   R0, R0, #1    ; R0 ready for forwarding
CMP   R1, #0        ; R1 ready for forwarding
```

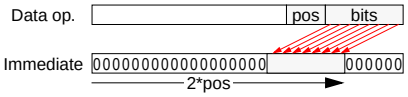
ARM Immediate representation

The original ARM instruction set uses 12 bits in its 32-bit instructions to represent immediate ('literal') operands. These come in two forms:

- `LDR/STR` use the value 'as is', *zero-extended* and then added to or subtracted from the base register according to a thirteenth bit.



- Data operations *zero-extend* an 8-bit field and right-rotate this to a position specified by the other 4 bits. 4 bits specify 16 positions so, to cover the full word, the rotation amount is doubled. This



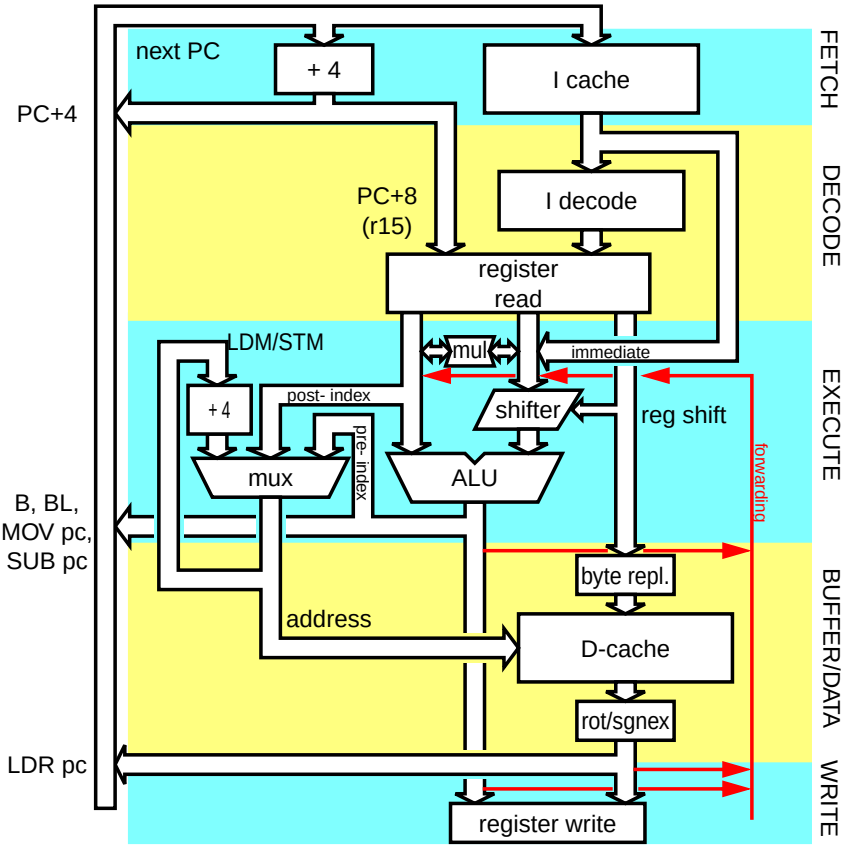
  places the 8-bit pattern in a selected even numbered bit position. The obtainable patterns form a generally a useful selection.

Thumb, with 16-bit instructions, uses much more limited immediate fields.

Thumb2 produces similar immediate fields to the original ARM using a 12-bit field within a 32-bit instruction although the coding is slightly more complex to offer slightly more flexibility.

ARM9TDMI pipeline

- ‘Harvard’ memory architecture
  - … as far as caches
  - parallel memory access
- Pipeline introduces more dependencies
- Alleviated with **forwarding** paths
- Multiplier:
  - 8 bits at a time
  - signed early termination



ARM9TDMI

The slide shows a datapath, arranged to try to show the pipelining. This has forced some compromises, such as the separation of the register read and write operations which (of course) affect the same physical registers.

The subdivision of the (long) execute stage shortens the critical path, meaning the clock frequency can increase.

There are more ports on the register bank than in the ARM7. This caused more design problems at circuit level (at that technology level) but simplified other operations. Examples:

- `STR R0, [R1, R2]` requires three reads; two cycles on ARM7
- `LDR R0, [R1], #4` requires two writes; ARM7 took two cycles

Forwarding paths alleviate most stalls.

Some (minor) extra functional units have appeared. There is an extra incrementer (‘+4’) to support LDM and STM. In the original ARM this was done using the same incrementer as the PC for fetch. Because there was a single memory bus fetches could not take place in parallel with loads or stores: on the ARM9 there is some opportunity for parallelism.

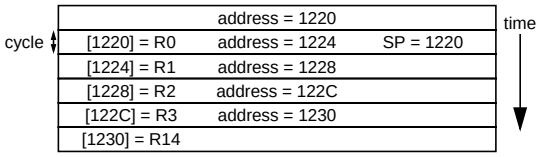
Interesting(?) consequence

The use of an *incrementer* (only) to supply addresses during LDM/STM has led to a feature of the ARM operation. The first address in any batch of loads or stores is always the lowest numbered address (here calculated by offsetting from the base address register using the ALU). Subsequent addresses occur in an ascending sequence. This is true even if (for example) pushing registers onto a *descending* stack.

Example: `STMFD SP!, {R0-R3, LR}  ; SP starts &1234`

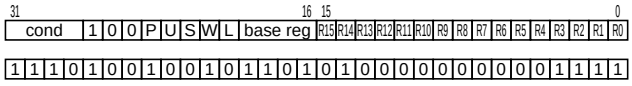
Starting the transfers is the most urgent process, so first count the registers in the list, multiply by 4 (shift) and subtract this from the SP.

Output this address with R0 on the data bus: in parallel increment the address register: in parallel, calculate the final value of SP and write it back. [In this case this is the same as the previous calculation but that is not always the case.]



Another example functional block

The example below shows the general encoding of ARM’s LDM/STM and the specific code for: `STMFD SP!, {R0-R3, LR}`



The encoding of the register list should be apparent. Slightly less obvious is the need for a **bit counting** unit to see how many registers there are in the list – and how far the base register may need to be offset.

Think about it a moment: it’s a non-trivial unit in itself.

What do the letters mean?

- T – ‘Thumb aware’, i.e. is able to decode Thumb instructions
  - This is compulsory in later architectures
- D – Debug: a JTAG tap for chip debugging is included
- M – Enhanced Multiplier (also 64-bit results)
- I – ICEBreaker: part of ARM’s in-circuit debug
  - Hardware registers can be compared with in real time to allow breakpoints and watchpoints on full-speed code

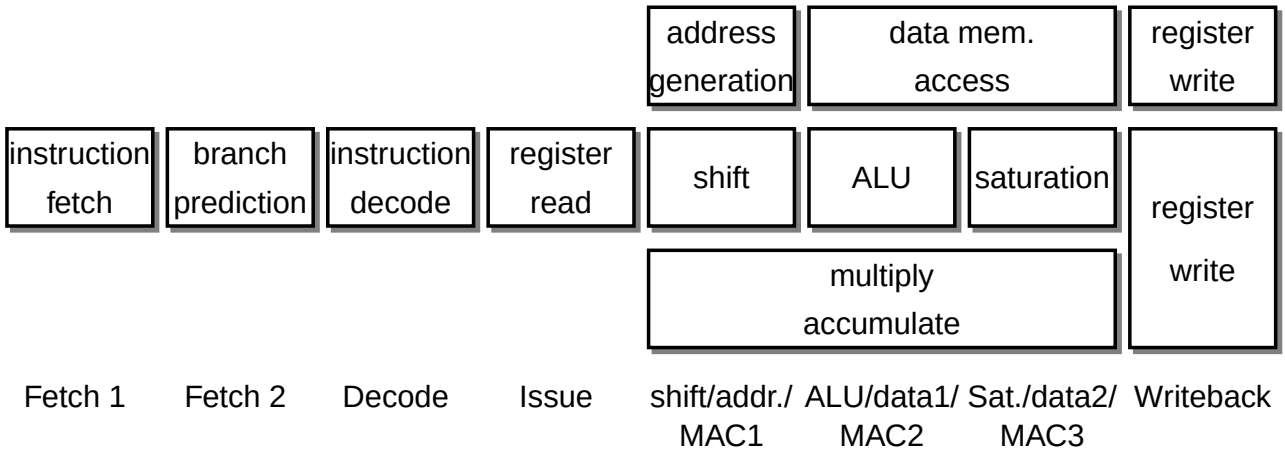
Other letters also appear.

- E – Enhanced (DSP) instruction set (new at that time)
- J –Jazelle mode

These are proprietary names, historic and only included in case you were wondering about them. Don’t memorise them!

ARM11 pipeline

- 8-stage pipeline



- Parallel ALU and data access (especially during LDM/STM)
- Additional adder for address generation
- Multi-stage multiplier: pipelined
- 'Hit under Miss' in 'data1' alleviates cache miss stalls

ARM 11 dependencies

- Data operations forward results
- Load operations impose an extra two cycle penalty (cache hit)
- Memory operations require their address registers early

Thus:

```
ADD     r2, r1, r0      ; produce R2
ADD     r4, r3, r2      ; consume R2 - no stall

LDR     r2, [r1]        ; load r2
ADD     r4, r3, r2      ; lose 2 cycles waiting

ADD     r2, r1, r0      ; produce R2
LDR     r4, [r2]        ; lose one cycle

LDR     r2, [r1]        ;
LDR     r4, [r2]        ; lose 3 cycles in total
```

- a few other dependencies may be seen from the pipeline figure

ARM 11 branch prediction

[Another **example**]

- Two-level dynamic branch prediction
  - constant offset branches
  - 128 entry, direct mapped ($addr_{[9:3]}$)
  - cost: 1 or 0 cycles if successful (taken/not taken)
- Static branch prediction
  - constant offset branches …
  - … that miss the dynamic predictor
  - cost: 4 cycles if successful
- Return stack
  - three entries
  - Pushes on `BL` or `BLX` (inc. `BLX Rn`)
  - Pops on: `{BX lr; MOV pc, lr; LDR pc, [sp], #n; LDMIA sp!, {…,pc}}`
  - cost: 4 cycles if successful

Jazelle

"Jazelle DBX" (Direct Bytecode eXecution) – originally just 'Jazelle' – is another instruction set offered on some ARM's, starting with an ARM7 enhancement. Jazelle is similar to Thumb in that it is a mode which alters the instruction decode process to interpret **Java byte code** directly.

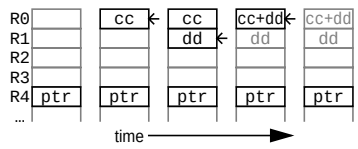
Full details of the operation of Jazelle are not public; here are some principles:

- Java byte codes form a machine instruction set but that 'machine' was designed to be implemented by software, i.e. as an *interpreter*. Because some Java byte codes specify complex operations, Jazelle does not implement the complete set in hardware, using software for some complex codes. As these codes are rare, the majority of the code is still executed at hardware speed.
- The Java Virtual Machine (JVM) has a stack-based architecture, quite dissimilar from ARM. To implement codes efficiently Jazelle uses some of the ARM's registers to implement an evaluation stack. Rather than shuffling data from register to register, this implies that a single byte code may translate *differently* on different cycles.

[Imagined example: "`dd = cc + dd`"]

```
Java code          ARM code
iload   3          ldr   r0, [r4, #12]   ; to R0
iload   4          ldr   r1, [r4, #16]   ; to R1
iadd               add   r0, r0, r1
istore  4          str   r0, [r4, #16]
```

Note: the 'top of stack' changes from r0 to r1 and back again.



The instruction translator must therefore contain some **state**, affected by its history. This is complex enough that it occupies **its own pipeline stage**. It would be possible to *bypass* this pipeline stage if Java decoding was not in use, reducing pipeline latency under certain circumstances.

In principle it would be possible to translate single byte codes into *sequences* of operations, stalling the prefetch whilst this occurred. ARM is silent on whether this has been done.

Cortex™

Three ‘flavours’:

- Cortex-A Series
  - Applications processors
  - ARM, Thumb and Thumb-2 instruction sets
  - ‘Performance’ oriented
- Cortex-R Series
  - Real-time embedded processors
  - ARM, Thumb, and Thumb-2 instruction sets
- Cortex-M Series
  - Microcontroller: deeply embedded/cost sensitive processors
  - Thumb-2 instruction set only
  - Small and low-power

Cortex™

“Cortex” is a branding name for the later series of ARMs

At time of writing (2013) *typical* family features are…

Cortex-A

- ARM, Thumb2
- Jazelle
- SIMD/DSP
- NEON
- Superscalar issue: in- or out-of-order
- Deep pipeline: 8-24 stages
- Fast(ish) clock (~1GHz)
- Security extensions
- (Some) 64-bit architecture
- 1-4 cores

Cortex-R

- ARM, Thumb2
- SIMD/DSP
- Hardware multiply/divide
- In- or out-of-order issue
- Medium pipeline: 8-11 stages
- Error Correction Codes (ECC)
- 1-2 cores

Cortex-M

- Thumb2 (occasional omissions)
- In-order issue
- Shallow pipeline: 2-3 stages (6 for M7: announced 2014)
- Single core

Example: Cortex-M3

First delivered Cortex-M

- Thumb2 instructions only
- New design
  - 3-stage pipeline
  - Branch prediction
  - Harvard core
- Small core
  - 0.12 mm^2 (90 nm technology)
- Low power (90 nm):
  - 32 µW/MHz
  - 1.25 DMIPS/MHz
  - ~40 000 MIPS/W
  - Sleep modes when not active reduce consumption to near 0
- No cache or MMU
  - Small Memory Protection Unit (MPU)

AArch32

From its origins there have been (arguably!) seven 'versions' of the ARM instruction set.

Each version has *added* some more features

- Extra instructions (lots of these)
- Thumb instruction set
- Operating modes, security features
- Coprocessors – particularly VFPU
- NEON

All these use the **same basic, underlying architecture** (as seen by the programmer)

AArch64

New architecture

- More, bigger registers (31 × 64-bit)
- Different address space (64-bit)
- Floating point
- Backwards compatible

New Instruction Set Architecture: i.e. the programmers' view is new

AArch64

The ARM 64-bit instruction set is more like a MIPS than an ARM. Many 'distinctive' ARM features are absent:

- 'Generality' of all registers
- Predication (condition code) on 'all' instructions
- Load/store multiple

Much more 'RISC' in character.

- 64-bit address space
- 32-bit instructions
- 31 64-bit general-purpose registers {X0-X30} + 'zero' register
  - Does not include PC, SP
  - 32- and 64-bit integer/address operations
- 32 128-bit registers
  - SIMD support
- Floating point support
- More 'specialist' function support
  - e.g. cryptography
- More system-level support in processor
  - Exception levels, virtual memory, virtualisation …
- Backward compatibility with 'AArch32'

Miscellaneous

DMIPS

MIPS (Million Instructions Per Second) measure the speed of a given microarchitecture. Of course, what is meant by 'instruction' varies between ISAs. Sometimes a figure is quoted in DMIPS which are Dhrystone MIPS. ('Dhrystone' was a benchmark program used once upon a time.) This allows a form of 'normalisation', the MIPS used referring to VAX[1] MIPS.

ARM instructions are typically a bit more 'powerful' than VAX MIPS so the DMIPS figure will be higher than the 'native' MIPS.

1. An obsolete ISA, once much used.

ARM 'modes'

The ARM model you will have seen (in lectures/labs.) to date has a single set of registers. This is not the complete story: some extra *physical* registers are overlaid into the bank of sixteen for operating system (O.S.) and exception support. Thus, for example, the O.S. has a different R13 (SP) from the user's R13, although only one is visible at a time.

The O.S. modes are also privileged, which means that the processor can so things to memory/devices which are denied to the user. This is for protection, to stop malicious or runaway user code from crashing the whole machine.

This is really outside the scope of this course. Come back in COMP22712 if you want to learn more.

Instruction emulation

Commercially it is often necessary to maintain backwards compatibility such that new, possibly enhanced, implementations are able to run older ISAs. This is certainly the case with architectures such as x86 and ARM. There can also be an issue of **forwards compatibility** where old processors are able to execute newly introduced instructions.

This requires that the space in the instruction set which is not used obeys some well defined behaviour. For example, ARM has a software exception – similar to an SVC – which is caused by any instruction which is "*undefined*". [In this case there are particular definitions of what is "undefined" which do not encompass every unused code.] These include coprocessor operations for a coprocessor which is not present. These can provide an **emulator trap**.

If an *undefined* instruction is encountered the trap can run software which examines the 'faulty' instruction and can potentially run a software emulation. As software is upgradeable, this means that old processors can (with appropriate support) execute code developed for newer ISA developments, albeit somewhat more slowly than hardware. This may be convenient, for example, for running floating point code on an implementation with no floating point hardware.