Computer Architecture

An embedded approach

Module 4

Processor Internals
(What's happening
in there?)

Contents

4.1 Internal bus architecture

4.2 Revisiting the ALU

4.3 Virtual memory

4.4 Cache

4.5 Co-processing

4.6 Floating Point Unit

4.7 SIMD/MMX

4.8 Other co-processors for embedded systems

A Programmers' Perspective

Assume we want to take two values and add them together;
A = B + C

In any CISC processor, we can probably always do this calculation without worrying about how complex the operation is in hardware. A, B and C might be registers, memory locations, accumulators, stack values...

However in a RISC processor, values A, B and C would probably have to be stored in registers before we begin.

Then, we would look in the CPU reference manual to find an ADD instruction we could use.

Just as in Section 3.3.4, we might find some different types of ADD that restrict what registers and values we can use...

A Programmers' Perspective

In most RISC processors, we can use any register for any part of the ADD operation:

```
ADD   R0, R0, R0
ADD   R15, R14, R1
ADD   R12, R12, R9
ADD   R12, R9, R9
```

This complete f exibility means something at an architectural level:

All registers must be connected equally to the buses that feed the ALU!

A Programmers' Perspective

In many RISC processors, also, the ADD takes only a single cycle to execute:

ADD   R1, R2, R3

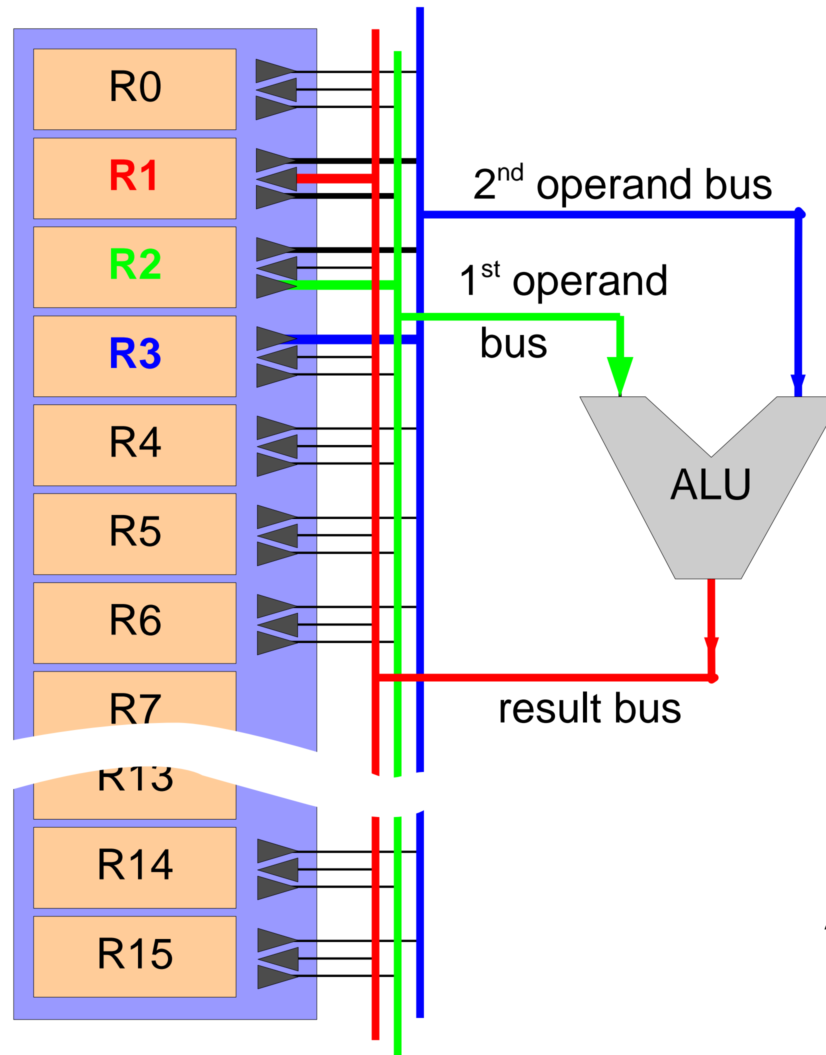
Two data items have to be moved to the ALU, the calculation needs to be performed, and then the result moved to a destination register...

If all this happens in a single cycle, it also means something special at an architectural level:

There are enough buses to convey two operands simultaneously to the ALU inputs, and at the same time, to convey the ALU result back to a destination register.

A Programmers' Perspective

The interconnection diagram probably looks something like this:

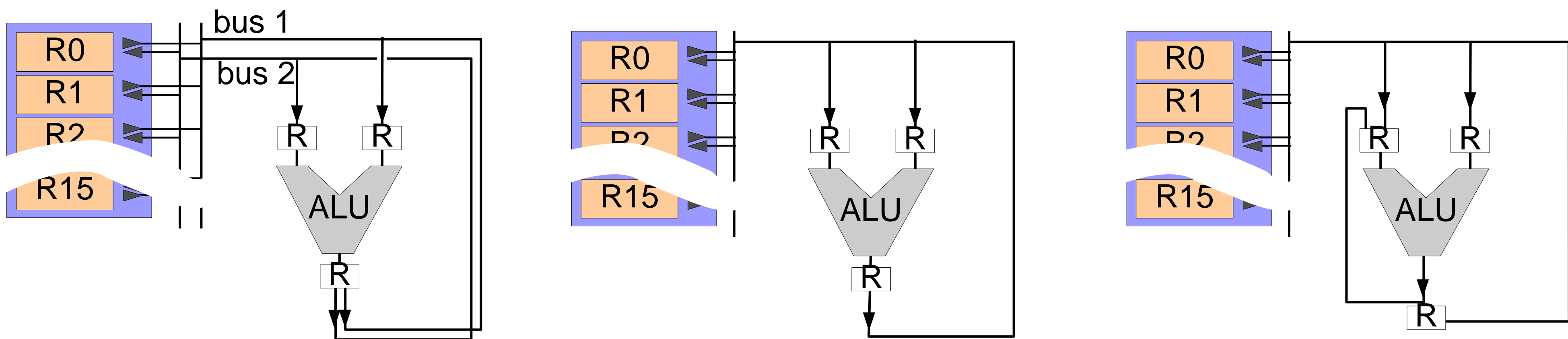


Three distinct buses needed, connected to almost everything...

ADD **R1**, **R2**, **R3**

A Programmers' Perspective

Consider the following alternatives. They all have the same banks of registers, the same ALU. However they have different numbers of buses.



i) The boxes labelled 'R' are registers – why do we need at least one register in these designs?

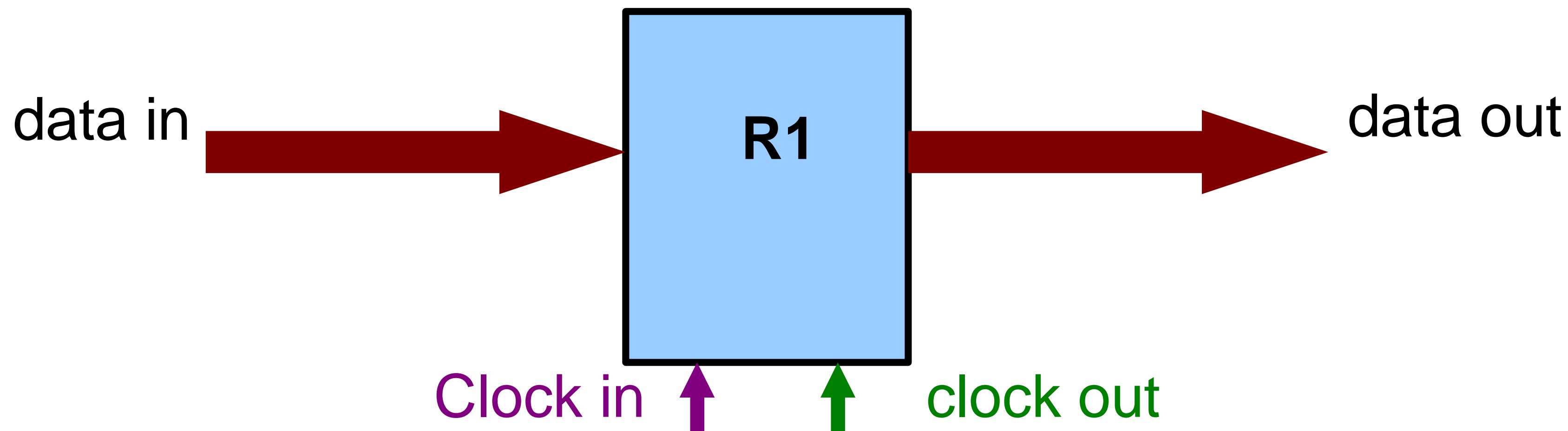
ii) None of these can execute R0 = R1 + R2 in a single cycle. How many cycles do they each take? Is R0 = R1 + R1 quicker?

A Hardware Perspective

The ALU takes values from one or two locations, performs an arithmetical operation, and writes the result to another location.

The result location may be the same as one of the input locations.

The locations are called registers. What is a register? It's simply a data latch:

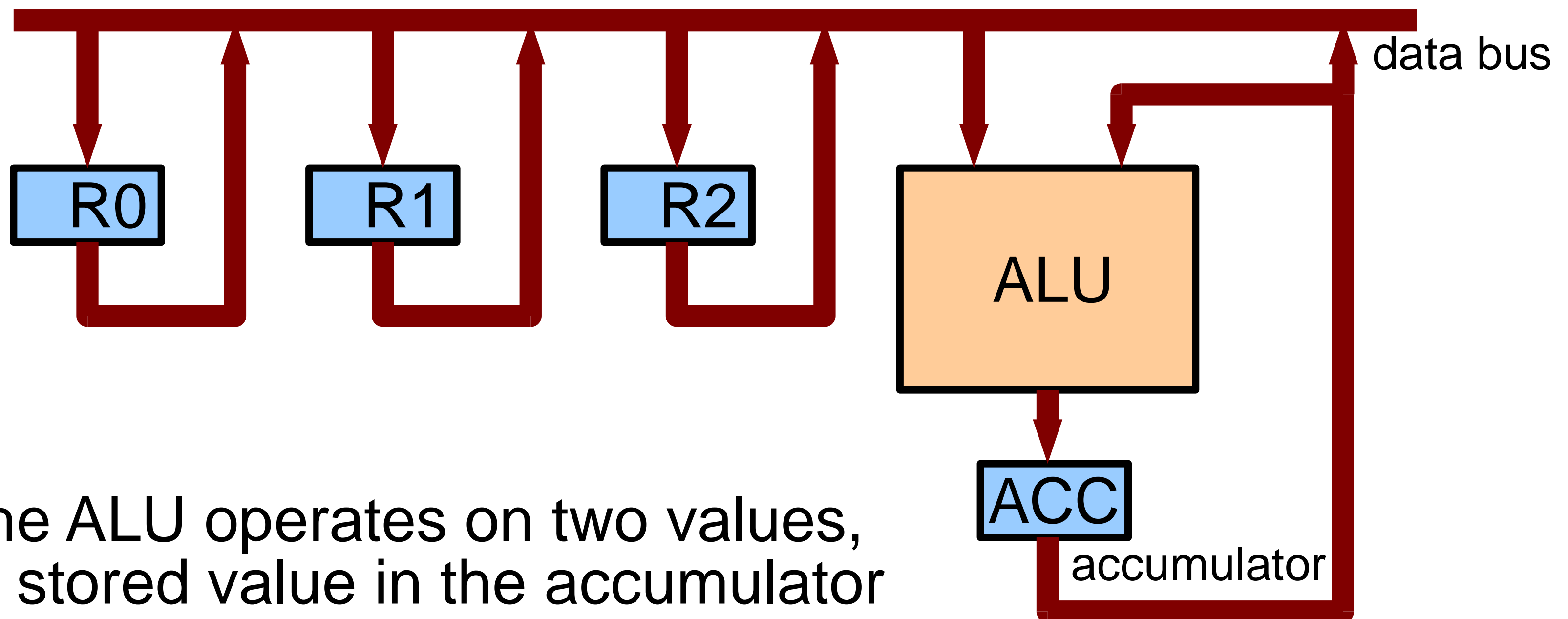


These two clocks operate out-of-phase to each other.

© 2011 Dr I. V. McLoughlin

A Hardware Perspective

Imagine a single bus architecture computer:



The ALU operates on two values,
(i) stored value in the accumulator
(ii) a value stored in an external register

The ALU is a combinational device: its output will be available a short time after its inputs have settled.

NOTE: The control logic is not shown

A Hardware Perspective

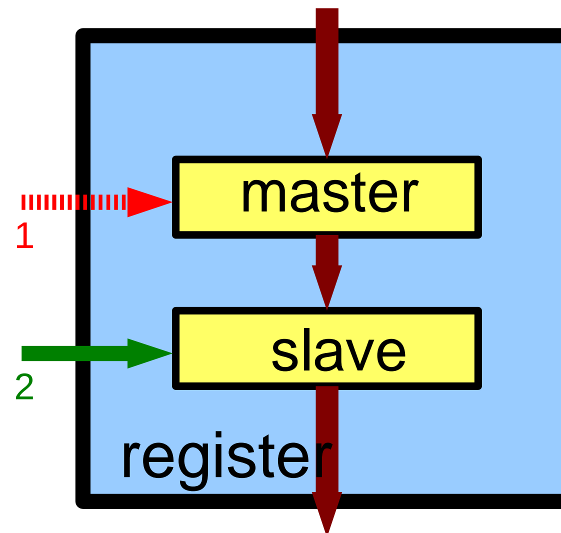
So the sequence of events is:

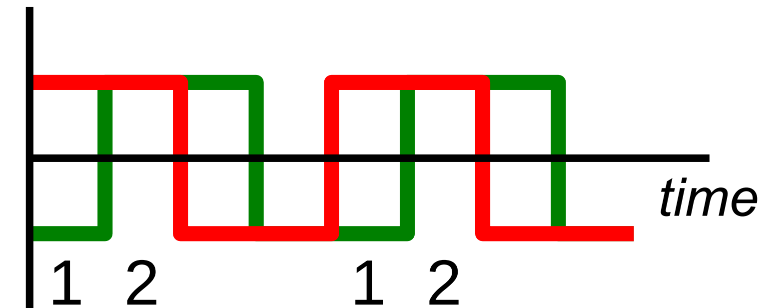
1. Place the first operand on the data bus
2. Output the second operand from the ACC
3. Wait for the ALU to perform its operation
4. Load result into ACC (unless result can go directly to the destination register)
5. Remove the first operand from the data bus
6. Remove the second operand at the ACC output
7. Output the result from ACC to another register

In practice we use a clock to ensure that the correct sequence is maintained. The clock needs to have two phases: $_1$ loads data into a register, and $_2$ outputs data from the register.

A Hardware Perspective

This is called a master-slave conf guration, using two f ip-f ops (per bit):



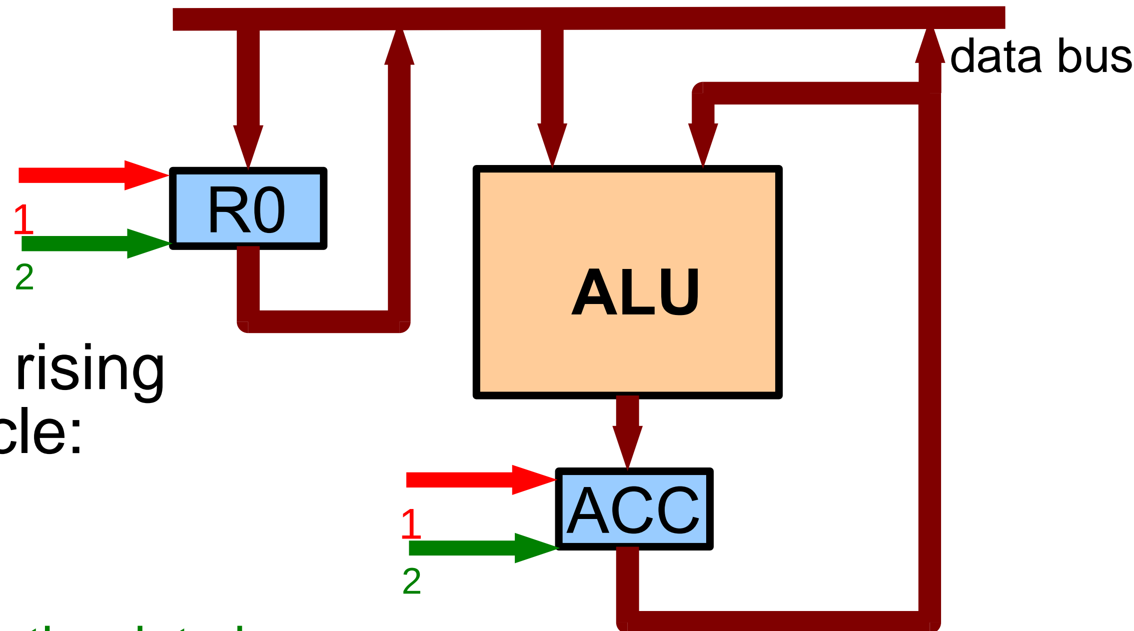


1. New data on the bus is latched by the master f ip-f op into the register on the rising edge of $_1$.

2. The stored data is latched as output from the register to the output bus by the slave f ip-f op on the rising edge of $_2$.

A Hardware Perspective

Looking again at the single bus system, identify which operations occur at the rising edge of which clock cycle:



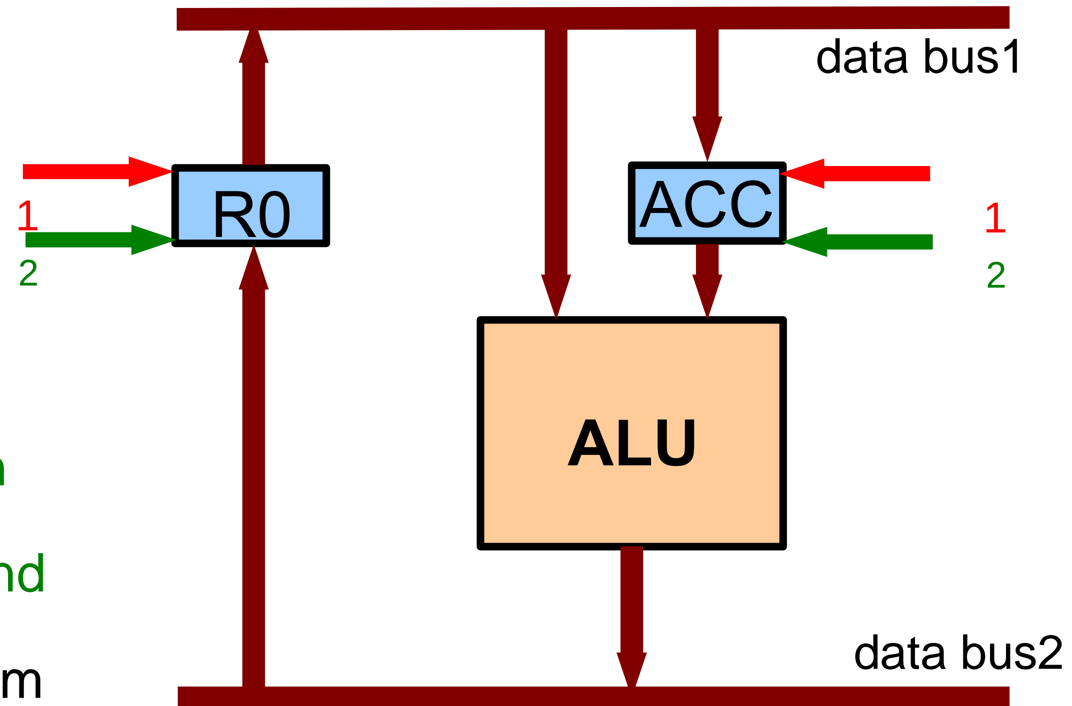
1. Place the f rst operand on the data bus
2. Output the second operand from the ACC
3. Wait for the ALU to perform its operation
4. Load the result into the ACC
5. Remove the f rst operand from the data bus
6. Remove the second operand at the ACC output
7. Output the result from ACC to another register

NOTE: The control logic is not shown

This operation needs 2 clock periods...
(*or 3 if we included having to load ACC f rst*)

A Hardware Perspective

Looking again at the ALU, we can propose a two-bus system to improve performance:



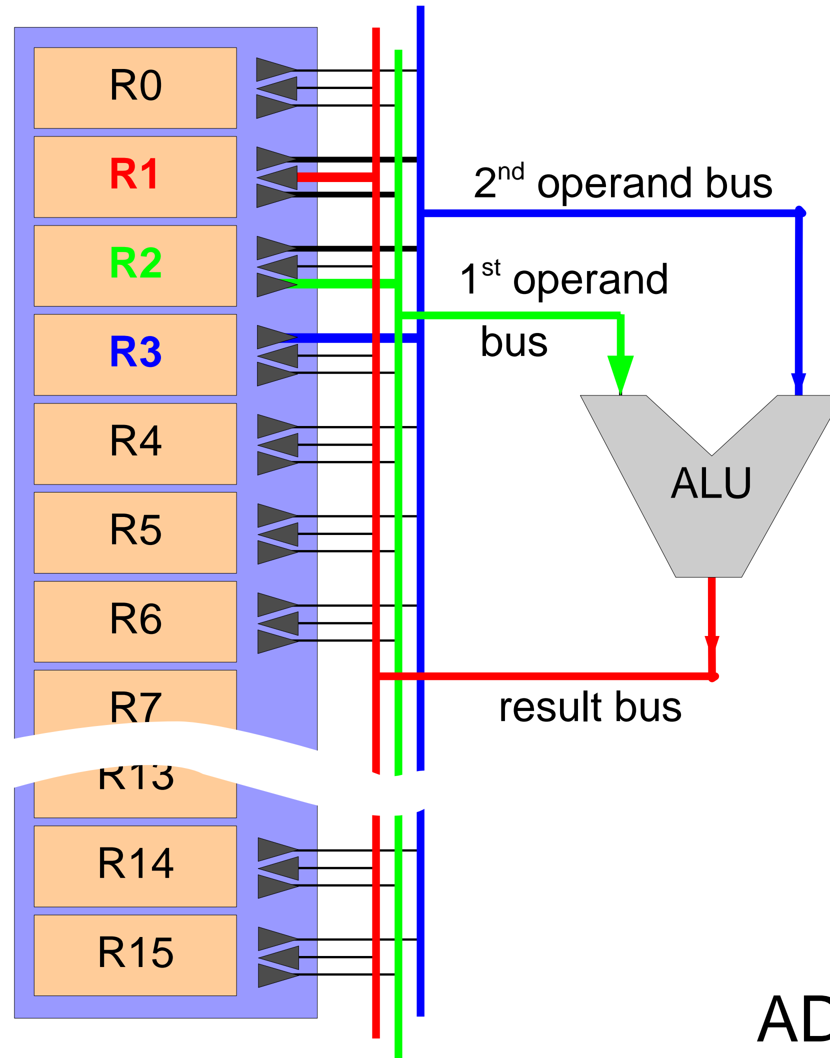
1. Place the first operand on data bus1
2. Output the second operand from the ACC to the ALU
3. Wait for the ALU to perform its operation
4. Load the result directly into R0

NOTE: The control logic is not shown

This operation needs only 1 clock cycle.

If we want to do any better, we need to go to a 3-bus architecture system (which we met a few pages ago):

A Hardware Perspective

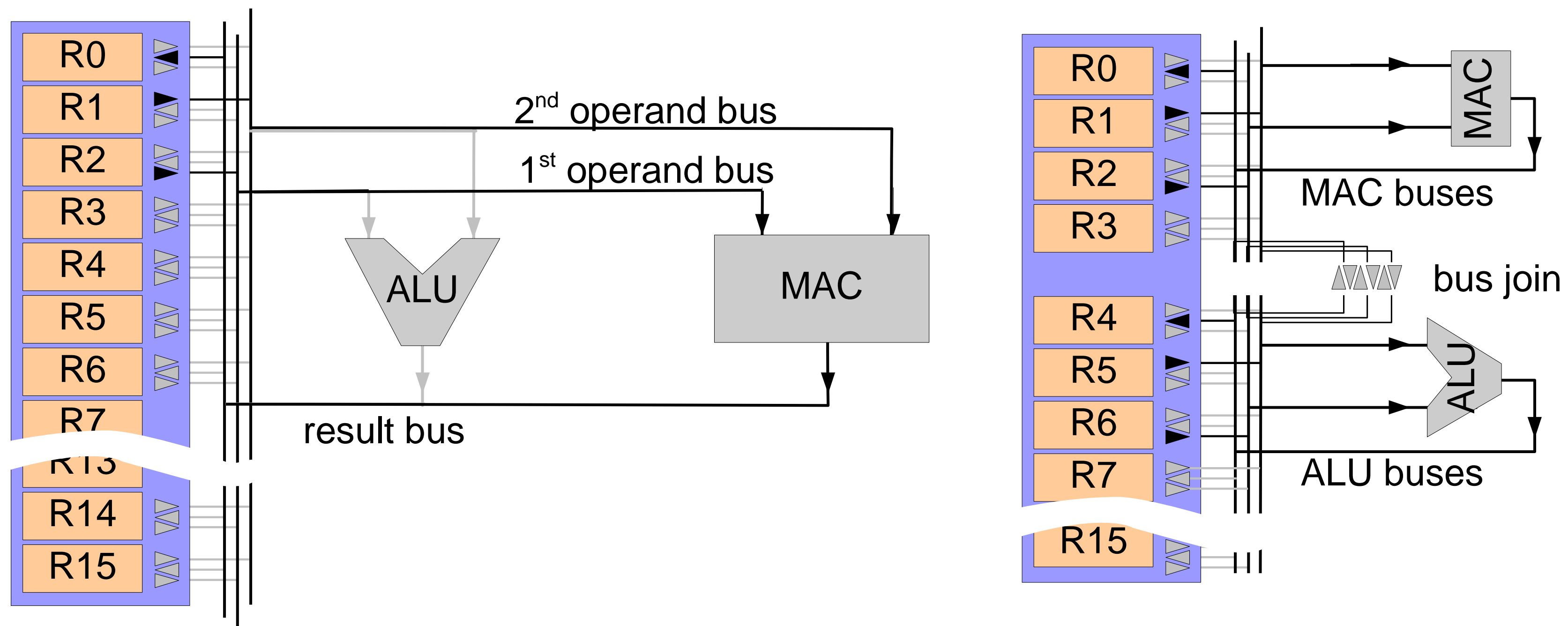


ADD R1, R2, R3

© 2011 Dr I. V. McLoughlin

A Hardware Perspective

However, we usually have more than just one functional unit. We could allow them to share the bus connections as in a typical RISC processor (left):

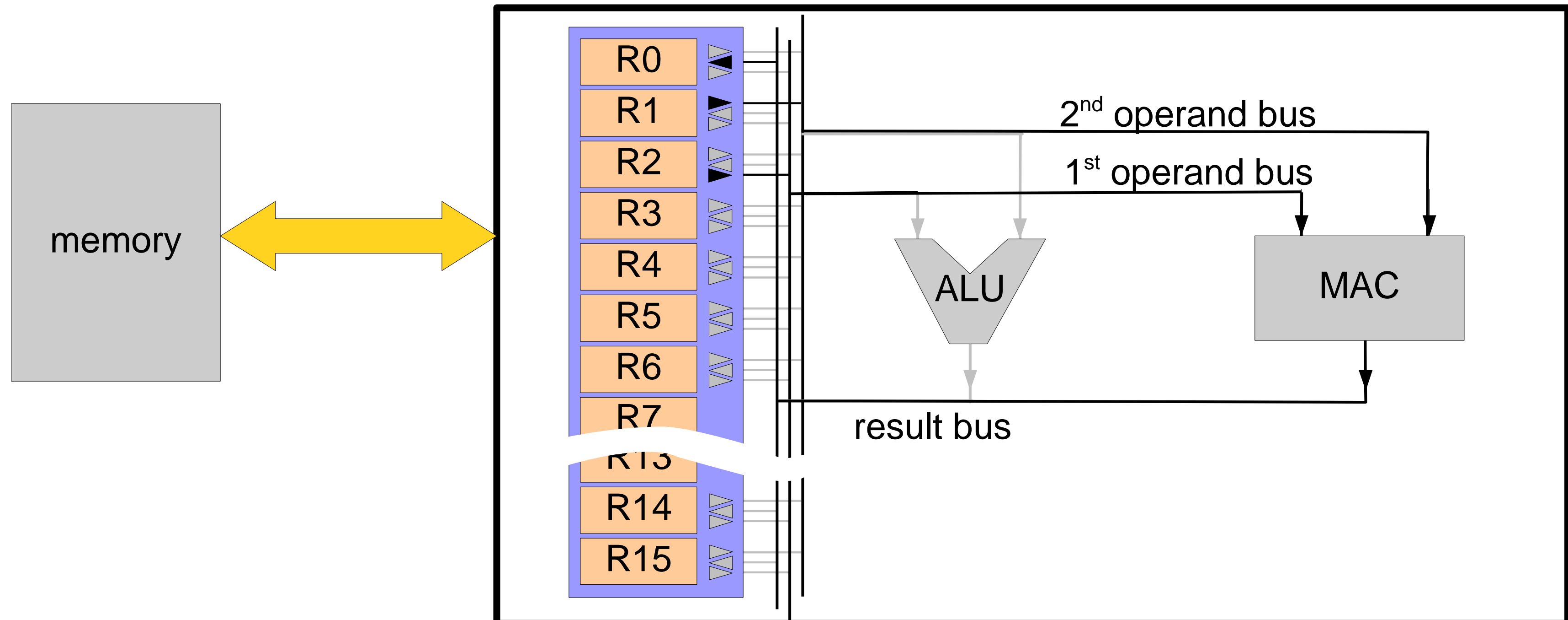


Alternatively, we could start splitting the buses into regions (right). This is used in some custom processors like the ADSP218x series DSP.

This makes for high speed, doing two operations per clock cycle, but is less f exible (and harder to program).

A Hardware Perspective

But this story does not end there...



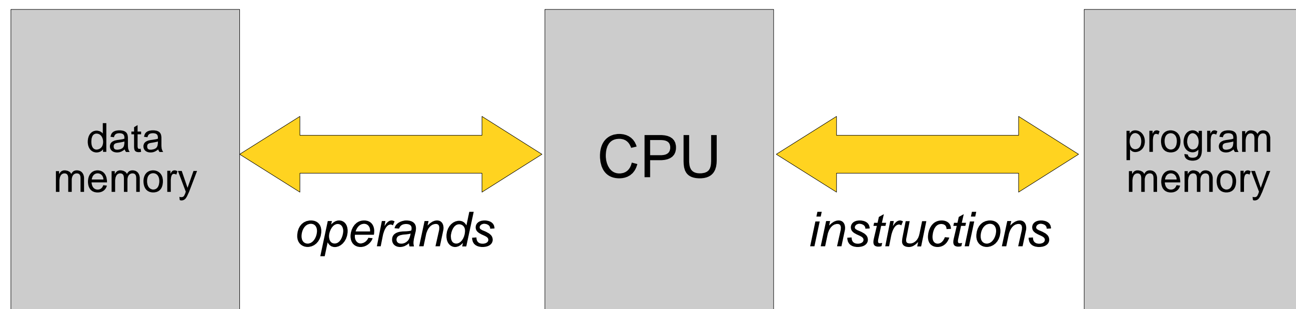
Both the data being acted upon, and the instructions which specify those actions have to come from memory.

Even if the operation takes only 1 cycle to perform, we still need to take 1 more cycle to fetch the instruction (and maybe more to fetch the operands if they are in memory).
Two operations in parallel would be even worse!!

A Hardware Perspective

One solution is to fetch the NEXT instruction at the same time as we process the PREVIOUS one. This is pipelining (see chapter 5).

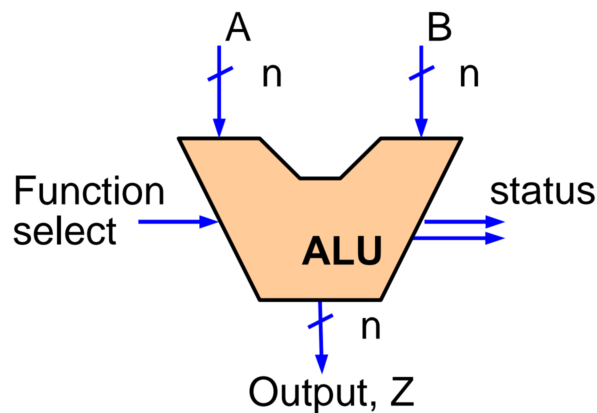
Another solution is to have two memory blocks. One which contains the instructions, and one which contains data: this is a Harvard architecture machine (as opposed to von Neumann machines which have shared memory).

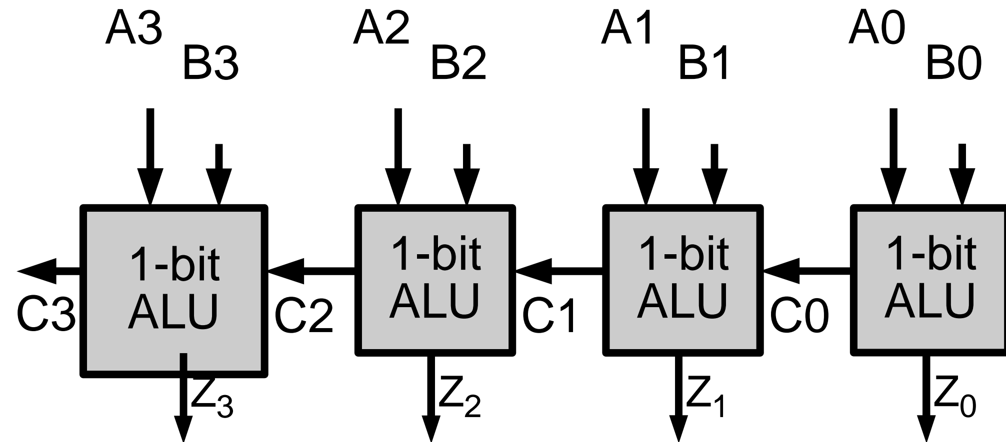


Or we could have single instructions that contain everything: parallel operations, immediate operands, data moves etc... We will explore this with EPIC/VLIW systems, in chapter 9.

Revisiting the ALU

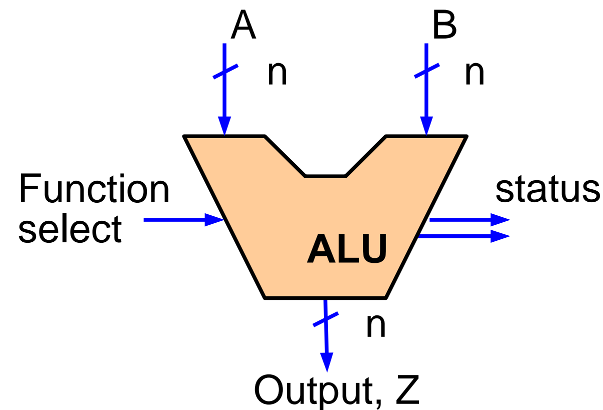
Remember that one ALU contains a number of bit slices. Logic operations such as AND, OR, NOR, all occur in parallel, but arithmetic operations ADD and SUB need to propagate their carries from lower bits up to higher bits.





It means that ADD and SUB are much slower operations... and the more bits in the numbers being added, the slower it becomes.

Revisiting the ALU



Usually the functions that can be selected are the following arithmetic and logical operations;

Z = A + B
Z= A AND B
Z= NOT A

Z = A – B
Z= A OR B
Z= A EOR B

However there are more possibilities in some ALUs...
NAND, NOR, DIV, MUL, remainder, multiply-accumulate

Revisiting the ALU

The status output ref ects information concerning the outcome of a calculation. The ARM ALU has 4 items of status:

N or negative ag
    the result of the last operation was negative

Z or zero ag
    the last operation resulted in a zero

C or carry ag
    the last operation generated a carry-out

V or over ow ag
    the last operation caused a sign change

Note that in the ARM, the programmer can decide whether the ALU leaves these f ags untouched, or changes them, after executing an instruction.

Revisiting the ALU

The following circuit diagram shows *one bit* of an ALU.
It ignores the status output, but shows how three functions can be selected:

carry in

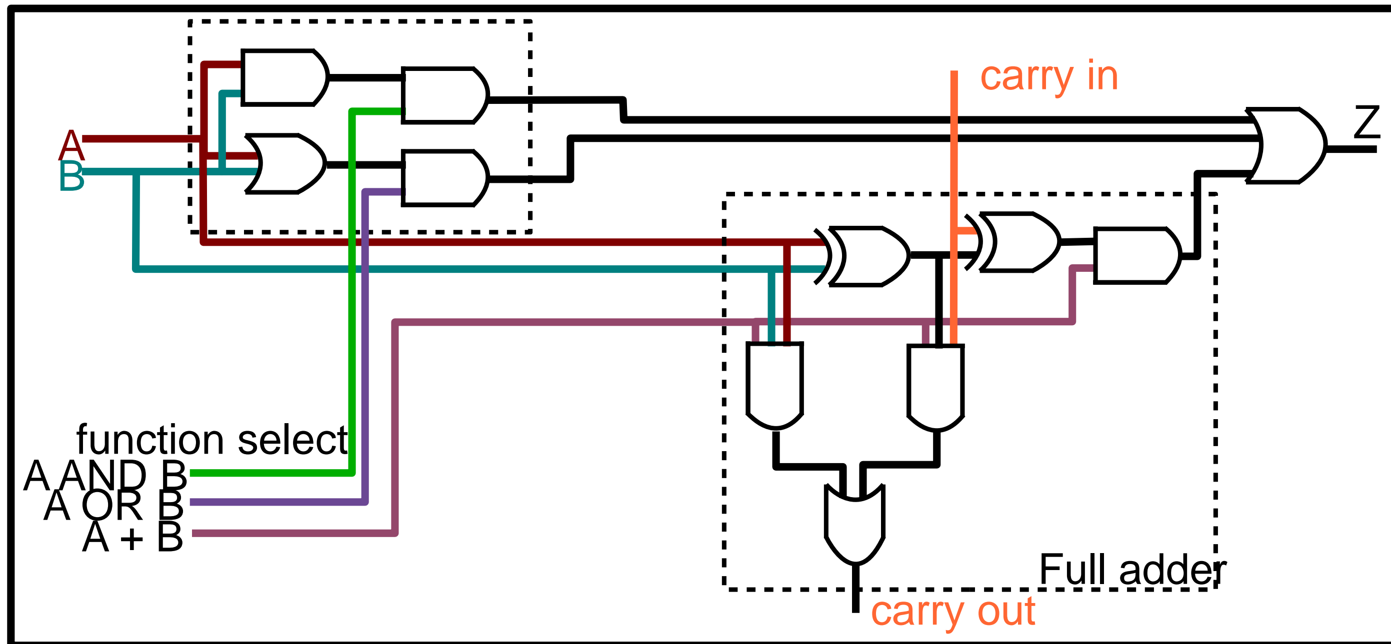
A
B

Z

function select
A AND B
A OR B
A + B

Full adder
carry out

Revisiting the ALU



This diagram is important: if we know the propagation delay of each gate (how long it takes for the gate output to stabilise once the inputs change), then we can work out:

- How much time one bit calculation takes for each operation.
- The worst-case time for one *n-bit* operation.
- The maximum clock speed of the ALU.

Refer to Box 4.1 for a worked example.

Computer Memory

In some ways, the story of computing over the past half century has been the story of a conflict between computer architects and computer programmers:

Computer architects make faster and more capable computers with more memory so your programs run faster...

Programmers then add extra features and capabilities which make the code bigger and slower...

One of the biggest battles has been over memory

Computer Memory

In computer architecture, there are several issues that we have with memory:

1. Too slow!
   Slow memory can 'starve' a CPU of instructions or data.

2. Too small!
   Can't load and execute large software applications.

Luckily, the standard computer architecture toolkit has some well-def ned ways to solve these problems (apart from “use more fast memory”, which is expensive):

Too slow –> use a cache

Too small –> use virtual memory

We will explore each of these in turn.

Virtual Memory

What is virtual memory?

This provides a very large space of memory that programs can 'see' and access. In reality, the physical memory is much smaller, with only the block (or blocks) of memory currently being used by the CPU, actually located in physical memory.
The remaining parts of the program and memory space are stored in slower/bigger storage (such as hard disc).

Blocks are loaded and unloaded as required.

A Memory Management Unit (MMU) handles *virtual memory* in a computer.

The technique was invented at Manchester University in 1962 bus is in extensive use today.

Virtual Memory

data bus

CPU

MMU

physical RAM

hard disc

32-bit address bus

20-bit address bus

32-bit address bus

An example MMU system for a 32-bit CPU: we will explore this further on the next page...

Virtual Memory

The CPU 'sees' a 32-bit address space (4 GBytes of memory), but the physical RAM is only 20-bits wide (1 MByte).
The MMU hides the small RAM from the CPU.

Memory is split into *pages*. Assuming pages are 256 kBytes in size, then main memory can hold only 4 pages at a time!
But the CPU can address up to 16384 pages.

The MMU's job is to load new pages into RAM, and store unused pages to hard disc.

If the CPU wants to read some data from a page that is not *loaded*, then the MMU first *retires* one page from RAM (stores it back to HD), and then loads in the requested page.

To know which page to retire, the MMU has to keep track of which pages are being used.

Virtual Memory

Inside the MMU:

virtual memory address from the CPU

page no.

line no.

physical RAM contents table

miss (page fault)

disc memory contents table

hit

The physical RAM address corresponding to the requested virtual memory location

The hard disc address that holds the required virtual memory page

Virtual Memory

Imagine the CPU wants to read from a particular address. The MMU splits this virtual address into a page and line.
e.g. if page size=20 then virtual address 29 is line 9, page 1.
Virtual address 44 would be line 4, page 2.

Next, the MMU looks in a physical RAM contents table for this page. This look-up table knows if requested page is already loaded in physical RAM, and if so, at what real RAM address.

If the page is already loaded, the value the CPU wants is simply retrieved from RAM at the page address listed in the table + the line number.
e.g. CPU reads from address 29: the MMU knows that page 1 is in RAM at address 0x0100. So it translates the read address to 0x0100+9 = 0x0109

page no.	loaded?	@ RAM address
16383	N	n/a
16382	N	n/a
	¦	¦
1	Y	0x0100
0	Y	0x0000

Virtual Memory

If the required page is not loaded, this is called a page fault, or miss. It means that the page number has to be sent to the disc memory contents table. This holds the hard disc address that the page resides at.

Before the new page is loaded from hard disc to physical RAM, space has to be made, by saving one of the pages that is already loaded, back to hard disc (and then updating the physical RAM contents table).

Different algorithms can be used to decide which page is retired back to the hard disc:

LRU (least recently used) - least recently used page retired
Can this cause any problems?

FIFO (First-in first-out) - the oldest loaded page is retired

© 2011 Dr I. V. McLoughlin

Virtual Memory

Speed of operation:

If the page needed is already loaded, reading from it is quick.

But a page fault (miss), means the CPU must wait until a page is retired, a new page is loaded, and finally the location is found and accessed in physical RAM. This is a stall.

A miss may be hundreds of times slower than a hit, so a good virtual memory strategy attempts to minimise the number of misses to maximise performance.

Virtual Memory

Fragmentation occurs when a program is allocated a number of fixed-size pages of memory, but does not completely fill the last page.

Every time that page is loaded into physical RAM, some unused space will be loaded and occupy the precious RAM!

So we need small pages!
But these are less efficient to load/retire, and mean bigger and slower contents tables..

Or, we can use variable length pages, called segments.

Virtual Memory

Ideally, segments should be able to grow dynamically during program execution (as more memory gets used).

Segments can be protected from each other.
For instance, program memory segments are executable, but read only. Data memory segments are read/write but not executable.

- Attempts to branch to a data segment should result in an error!

- Attempts to overwrite a program memory segment should also f ag an error!

These are really Operating System issues – we are more interested in the MMU hardware operations...

Virtual Memory

Segmented memory spaces can be more eff cient than the earlier paged systems because they do not suffer from internal fragmentation.
However they are more complicated because we need to keep track of the size and contents of each segment.

Unfortunately they also suffer from *external fragmentation*:

© 2011 Dr I. V. McLoughlin

Virtual Memory

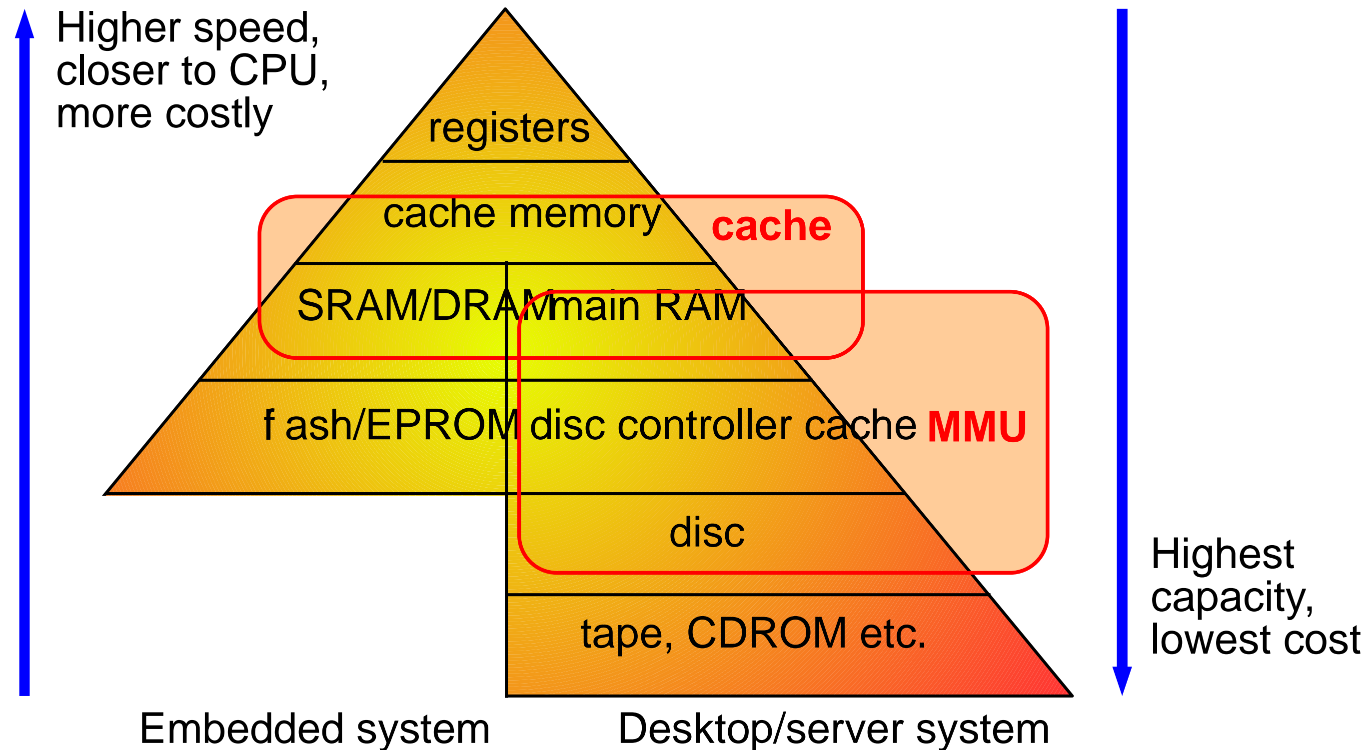
To prevent external fragmentation, the memory area must be compacted periodically:

Unfortunately, compaction takes some time, so it is usually performed only when necessary.

The world has many segment management algorithms, but all of them need to track used and unused portions of memory.

Memory in Computer Systems

Remember the memory hierarchy from Chapter 3?



We've just looked at the MMU – it can be slow but provides a large virtual memory space to accommodate big programs.

Next we focus on making memory faster with the cache.

Cache Memory

The highest level caches (close to the CPU), are usually implemented as fast on-chip memory.
These tend to be small (up to ~100k for ARM CPUs).

The Pentium Pro was an interesting case with two silicon chips in one package: a 256k cache, and a Pentium CPU.

Unfortunately, there were many manufacturing issues...

Cache Memory

Split caches can be used separately for data and instructions, especially for Harvard architecture processors.

Similarly to virtual memory, a cache miss is when the required data is not in the cache, and has to be fetched from slower memory.
Of course, some data has to be retired f rst, and possible some compaction take place.

The hit ratio is the percentage of wanted memory locations that are found in the cache. The key to good cache design is to maximise the hit ratio.

Let's consider different forms of cache organisation...

Cache Memory

Direct cache: Assume the cache has 2^n entries, or lines.

Each cache location can hold one line of data from memory.

Each memory location corresponds to a f xed cache location (and as the cache is much smaller than the memory, each cache location must map to to many memory locations).

So when the cache needs to check whether it holds a particular memory address, it just needs to look in one cache location, based on the address, for a matching tag.

The cache line is taken from the lowest *n* bits of the memory address:

© 2011 Dr I. V. McLoughlin

Cache Memory

Direct caches actually contains a number of fields.
A dirty/clean flag indicates if a value has been changed (but not yet stored back to main memory).
A valid bit indicates if the field is occupied or not.
The tag entry indicates which of the possible memory locations is actually being cached.

	valid	dirty	tag	data
line 1023	✔	☹	0000	0000 2001
line 1022	✔	☺	0001	FFFF FFF1
line 2	✔	☺	0100	0000 0051
line 1	✗	☺	XXXX	XXXX XXXX
line 0	✔	☹	0000	1A23 2351

© 2011 Dr I. V. McLoughlin

Cache Memory

So the direct cache operation algorithm is:

TO READ - split the required address into TAG and LINE.

Check the cache at the LINE location and see if the TAG entry matches the requested one.

If it does, read the value from the cache.
If the TAGs do not match then look at the dirty f ag.
If this is set, f rst store the current cache entry on that line back to main memory. Then read the main memory value at the required address into that cache line.

Clear the dirty f ag, set the valid f ag and update the TAG entry.

Cache Memory

TO WRITE - there is a choice:

write through writes the value into the cache line (first storing any dirty entry that was already there), and also writes the value into main memory.
write back does not store into main memory (this will only happen next time another memory location needs to use the same line).
write deferred allows the write into the cache, and some time later (when there is time available, the cache line is written back to main memory).

Whenever the cache value is written to main memory, the dirty flag must be cleared.

With write through, if the memory location is not already in the cache, it is possible to directly store the data to memory, bypassing the cache.

Cache Memory

Set associative caches:
The problem with the direct cache was that address locations 0, 1024, 2048, 3072 ... etc. all compete for one cache line.
To solve this, an *n*-way set-associative cache allows *n* entries to each line:

	valid	dirty	tag	data	valid	dirty	tag	data
line 1023			0000	0000 2001			0015	0110 2409
line 1022			0001	FFFF FFF1			0002	0000 0003
line 2			0100	0000 0051			XXXX	XXXX XXXX
line 1			XXXX	XXXX XXXX			0006	FFF1 3060
line 0			0000	1A23 2351			0004	4A93 B35F

Cache Memory

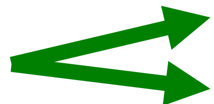
Set associative caches:

In our example 2-way set associative cache, there are two possible locations where we can cache any main memory address. Which one do we choose? Again there are a choice of algorithms. One good algorithm is LRU (least recently used) that we discussed under the MMU section (good means better hit/miss ratio).. see later

Time taken for a hit: Test for hit + retrieve value from cache

Time taken for a miss: Test for hit
Run LRU algorithm
Check for dirty f ag, and if necessary,
save current cache contents to memory then
Load wanted value from memory to cache
Retrieve value from cache.

These operations are very slow



Cache Memory

Full associative caches:
Any memory location can be mapped into any cache location.
In this case the cache TAG holds the full address of its content.

The problem is that when the cache is asked to retrieve a location, every cache entry TAG must be checked.
In the direct case, only one TAG needed to be checked.
In the *n*-way set associative cache, only *n* TAGs had to be checked.

So although the chances of getting a good hit/miss ratio are better, the operation of the cache itself is slower because of the increased checking.

Cache Memory

Full associative caches:
Some caches actually read blocks of memory rather than just single memory locations. In this case, the TAG is the start address of the block, and the cache controller knows that *m* consecutive memory locations are cached in one cache line.

Cache coherency: ensuring that all copies of a memory location in caches hold the same value. This is particularly important for multiprocessor systems when a value written to one CPUs cache may be needed by another CPU.

How does the second CPU know that its cached value is no longer the latest value?

Cache Memory

Replacement Algorithms - which cache location is replaced when there are several possibilities to choose from?

LRU scales with the size of the cache, and performs reasonably

FIFO replace the location that has been longest in the cache. Is easy to implement in hardware, and also performs reasonably well

LFU replace the least frequently used location. This needs each cache entry to have a counter & circuitry to compare the counters.

Random very easy to implement in hardware; just pick a random location. Surprisingly this technique performs quite well.

caches must be FAST, so these need to be implemented in hardware

Cache Memory

Cache performance:

Variables used within a program are usually stored consecutively in memory (this is the principle of spatial locality). The levels of procedure nesting are usually limited, so only a few sets of variables are in use at any one time (this is temporal locality: *see Section 4.4.4 of the book for an explanation of locality*).

If cache location M_1 has access time T_1 for a hit, but for a miss we need to transfer word M_2 from main memory into M_1, with transfer time T_2 and hit rate, H=no.cache hits/no.requests then overall access time:

$$T_S = H \times T_1 + (1 - H)(T_1 + T_2) = T_1 + (1 - H)T_2$$

Cache Memory

Cache design is a compromise between speed, size and cost:

Decide average memory access time required.
Work out the hit rate needed.
Choose cache size to give required hit rate.
Check average cost of memory per bit is not too expensive.

For your information, a good cache would probably have a hit ratio of over 0.75.

Some processors have fast internal memory instead.
The ADSP2181 has 80k of on-chip memory which can all be accessed within a single instruction cycle, so no cache may be needed... some embedded CPUs have similar arrangements.

Refer to Section 4.4.6 in the book for cache design and performance analysis.

Co-processing

Sometimes processors contain a co-processor to handle specialised tasks. These tasks might be operations that are diff cult or slow on a general purpose CPU.

The co-processor might save energy, time or just free up the CPU for other tasks.

Here are some common ones (roughly arranged with the most common at the top):

- Floating point unit
- Graphics processing unit
- Network or input/output processing element
- Graphics processing unit
- Cryptography unit
- Audio processing unit
- Wireless unit

Floating point unit

The Pentium FPU is basically unchanged since it appeared in about 1986 as the 80387, which was a separate co-pro for the 80386. Today it's just included as standard in the Core CPUs and beyond.

In operation, the CPU loads operands into special registers which are shared between the main CPU and the FPU, and then the FPU is activated.
Some time later, the FPU returns the result to the special register area, and optionally informs the CPU through an interrupt. Most modern processors include the FPU inside an execution pipeline.

FPUs can't access data directly, only operate on what the CPU
passes to those special purpose registers (long enough to hold IEEE 754 double precision numbers).

Floating point unit

Many embedded CPUs, especially for smaller or low-power systems, do not have a dedicated f oating point unit (FPU).

Instead, they use a software f oating point emulator (FPE) which can be tens or even hundreds of times slower than an FPU.

Therefore, if you write code for such systems, try not to use f oat or double data types – stick to int and short if possible.

SIMD/MMX/SSE

The history of these devices has really been to accelerate some of the types of data more commonly used in desktop computers:

MMX: multimedia extensions
SSE: streaming SIMD extensions from Intel
3DNow!: AMDs version of MMX/SSE

Although they still exist, the original reason was largely for speeding up graphics handling. However these have largely been superseded by graphics processing units (GPU).

Still, sometimes, great performance gains can be found by rewriting some intensive code routines to use these units.

Co-processing in embedded systems

Consider the following co-processors available on ARM computers:

Jazelle	for speeding up Java processing
NEON	advanced SIMD accelerator similar to SSE/3DNow!
VFP	vector f oating point co-processor

Cryptographic engines are also becoming more popular on embedded CPUs as the need for security increases with the more and more important role that embedded systems are gaining in our lives.

One very capable system is the integration of a small FPGA with an ARM processor. The FPGA can be conf gured as almost any co-processor and used by the ARM to accelerate various types of processing.

© 2011 Dr I. V. McLoughlin