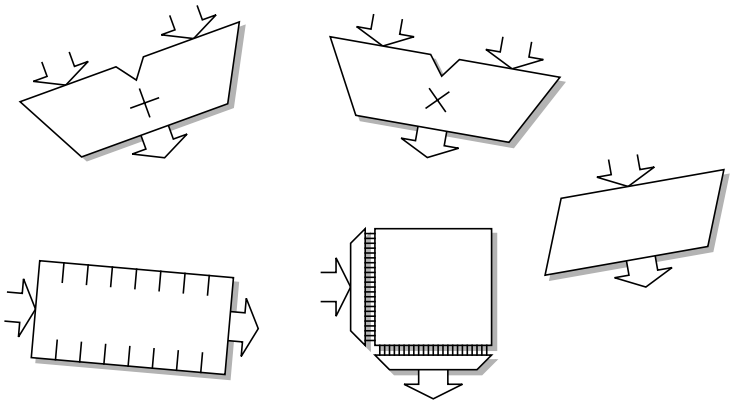
Microarchitectural structures

The purpose of this lecture is to examine some (more) of the structural blocks which commonly comprise processors … and other VLSI units.

- This is not a comprehensive list (of course)
  - Does cover many of the simpler common structures

Examples:

- Arithmetic & logic
  - Adders
  - Multipliers
  - Shifters
- Buffering and storage
  - FIFOs
  - RAM
- Prefetch stage – a more complex example
  - Including simple a branch predictor



Microarchitectural structures

It is not possible to list **all** the structures which may be required when designing ASICs. This lecture covers a selection of the more general structures.

Some of these – such as adders – you will have met before; others may be new.

Some of these – at least in some form – are represented in Verilog and may be synthesised trivially.

Here the adder is again a good example. If you write "`A + B`" you can expect an adder of the appropriate size to be synthesised. However it is still useful to have some appreciation of the underlying technology because this will influence the speed and size of the resultant part. CAD tools go a long way to satisfy the designer's constraints but the task can be made easier by specifying something 'sensible' in the first place.

A typical example is multiplication. There are lots of choices which can be made depending on hardware budget, latency, throughput etc. Some Verilog synthesisers will not synthesise a simple '`*`' operation for this reason. The Xilinx FPGA tool we use *will* build multipliers (naïvely) but allows user input to let it divide the operation into pipeline stages, for example.

- -

Register file

The implementation of a register file (a.k.a. "register bank") depends on several considerations, including the technology available. The decision will be influenced by the number of physical registers implemented: a small file may fit comfortably in D-type flip-flops; a large file may be better in RAM. [Note that there could be more physical registers than architectural ones due to 'advanced' implementaion techniques such as *register renaming* or *hyperthreading*.]

Another influence will be the number of simultaneous reads and writes which can be performed. This will be influenced by both the ISA and the implementation. Example: the vast majority or ARM instructions have no more than two register reads and one write. However, what about these?

```
STR   R0, [R1, R2]
LDR   R3, [SP], #4
```

Priority encoder

To fill in this column, here is a worked example of the sort of structure that may crop up in a number of different hardware units.

Function

If at least one from a set of inputs is asserted, choose the one with the highest priority. The priority may be fixed (as here) or it may be defined dynamically (e.g. promote an input not serviced 'recently').

Example uses

- Choose which of a set of possible interrupt requests to service next
- Choose an input packet for routeing around a network

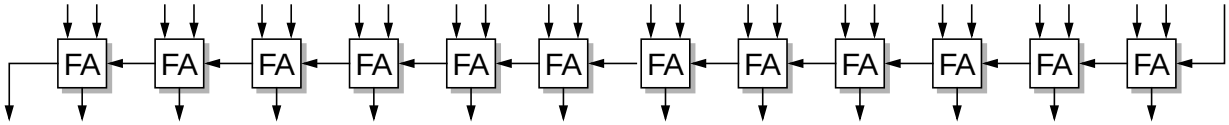
Implementation

```
module priority (input  I3, I2, I1, I0,
                 output req,
                 output reg [1:0] choice);
begin

always @ (I3, I2, I1, I0);       // Prioritize
  if (I3)      choice = 2'b11;
  else if (I2) choice = 2'b10;
  else if (I1) choice = 2'b01;
  else         choice = 2'b00;

assign req = I3 | I2 | I1 | I0;  // Any input?

end
endmodule
```

The very enthusiastic student may want to read up on the Verilog '`casex`' statement for a 'neater' way of expressing such a construct.

Adders

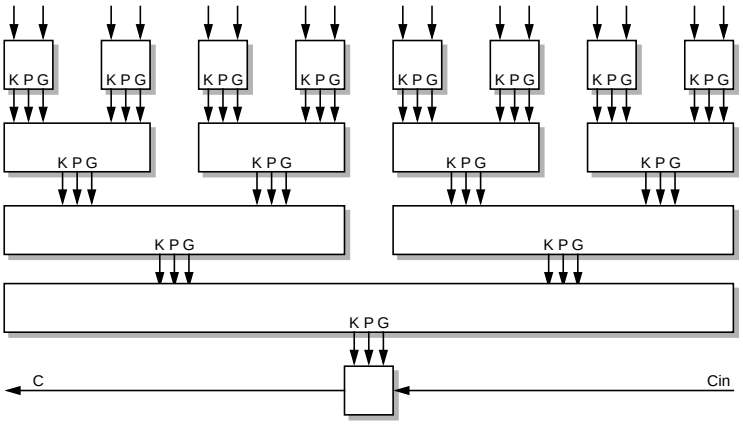
Ripple carry



Simple small slow (long critical path)

Carry look-ahead

Can decide what carry should be in logarithmic time



A	B	C	
0	0	0	Kill
0	1	C_{in}	Propagate
1	0	C_{in}	Propagate
1	1	1	Generate

S_1	S_0	
K	?	K
P	K	K
P	P	P
P	G	G
G	?	G

{K, P, G} – only need any two

Can, sensibly, evaluate for 3 or 4 bits at any stage: thus can propagate across 64 bits >10× faster.

Adders

From a HDL an adder is a basic block and the logic synthesizer will usually find an appropriate implementation. The following descriptions cover some VLSI alternatives.

Ripple-carry adder

The ripple carry adder is simple and compact. Its operation reflects the way a human might perform an addition: one bit at a time starting at the least significant end. Although all the inputs may arrive simultaneously the evaluation of the more significant bits will not be complete until their carry input arrives.

The carry is evaluated sequentially and 'ripples' through the bits which results in a long *critical path*. Although, in many cases the carry will not matter – for example when adding two '0' bits locally a carry cannot be propagated because the eventual result can only be '00' or '01' – in some cases the carry may travel across the whole word and this sets the minimum clock period.

An N-bit ripple carry adder takes a time O(N).

Carry look-ahead adder

Before a carry it can be determined that each bit position will output one of three carry states.

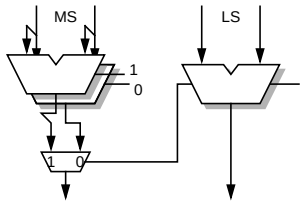
- Kill – inputs are {'0', '0'} so the carry out must be '0'
- Propagate – inputs are different so carry out will be carry in
- Generate – inputs are {'1', '1'} so the carry out must be '1'

This information can be evaluated quickly. It can then be combined from adjacent bits to determine the same properties over more bit positions. Repeat!

The result is that a set adjacent of bits can be 'ready' to propagate an incoming carry across their entire width with a single gate, rather than 'rippling' through the set of single bit adders. The benefits increase as the adder becomes wider as the delay is O(logN) where the base of the log. is the fan-in of the look-ahead units. There is considerable hardware investment to achieve this though.

Refined examples include: Manchester carry chain, Brent-Kung adders and Kogge-Stone adders

Carry-select adder



The carry-select adder duplicates a section of the adder and speculates on what the input carry will be. In the figure here the less significant part is a 'simple' adder (could be ripple carry, CLA etc.) and the more significant part is duplicated, one copy assuming a carry input of '0', the other assuming a '1'.

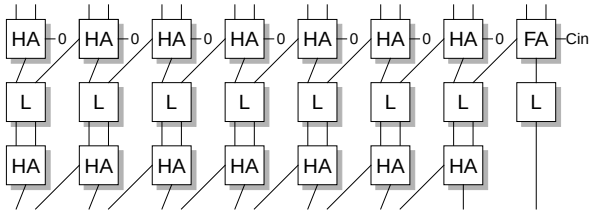
When the carry is known the appropriate output is *selected* using a multiplexer. The delay of a multiplexer is quite short.

The structure allows the two 'halves' of the word to be evaluated largely in parallel although there is a significant hardware overhead. The division into 'halves' is arbitrary and more sections can be made, tailoring their widths appropriately. The speed up depends on this choice, and the internal architecture of the adder blocks.

Carry-save adder

A different approach, a carry-save adder can achieve *throughput* comparable (or greater) than the adders above at low cost, providing that the addition can be pipelined and a *long latency* is allowed. This is typical in operations which require many successive additions, multiplication being one simple example.

A carry-save adder is basically like a ripple carry adder but the 'ripple' is delayed in time into a subsequent clock cycle.



This is not very useful for single additions; for repeated additions, replace some half-adders with full adders and each extra stage performs an entire addition.

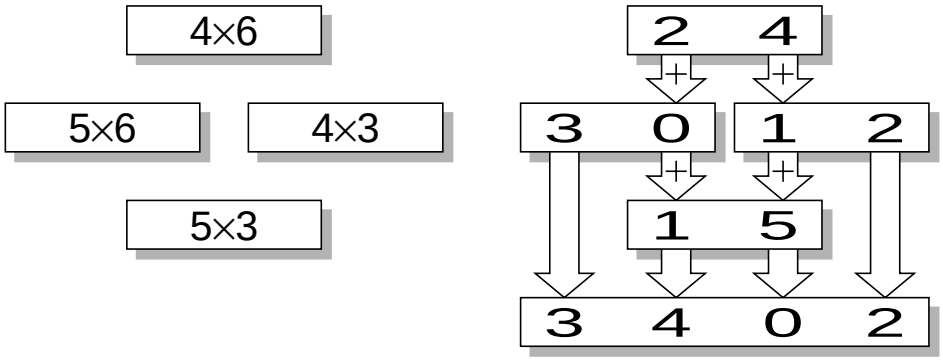
Multipliers

- Multiply one input by each **digit** of the other
  - easy in binary
- **Shift** the partial products into the correct orientation
- **Add** the justified partial products

```
    5 4
  × 6 3
  -----
    1 6 2   =  50×3  +  4×3
  3 2 4 0   =  50×60 + 4×60
  -------
  3 4 0 2
```

Plenty of opportunity for parallelism: e.g. can evaluate several partial products simultaneously.

Also small multiplier blocks can be composed to make larger multipliers



This technique is used, for example in FPGA synthesis where several small (pre-optimised) hardware multipliers are available for composition as required.

Pipelining is also possible if appropriate.

Multipliers

'Long' multiplication is done by *shifting* partial products by powers of the base:

```
                    5 6 7 8
                  × 1 2 3 4
                  ---------
5 6 7 8 × 4 ⇒       2 2 7 1 2
5 6 7 8 × 3 ⇒     1 7 0 3 4 0
5 6 7 8 × 2 ⇒   1 1 3 5 6 0 0
5 6 7 8 × 1 ⇒   5 6 7 8 0 0 0
                ---------------
                7 0 0 6 6 5 2
```

The partials can be added, appropriately justified, to achieve the result with ('number of digits' - 1) additions.

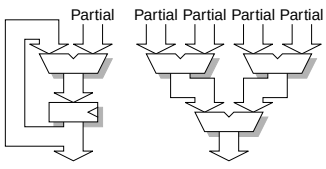
[May be easier to think of 'number of digits' additions, starting with a '0'; this can then support multiply-accumulate by not starting with a zero.]

Binary multiplication is easy:

`0 × something = 0 : 1 × something = something`

This should look like a familiar Boolean operation, namely: AND

Thus an N-bit binary multiplier can produce an answer with N additions.



The additions do not need to be in any particular order. They can be done *iteratively*, using an adder-accumulator or in parallel using multiple adders, say in a tree-structure, or some combination of structures. There is a cost/performance trade-off to consider here.

The adders do not need to resolve the carry propagation on every cycle: see, for example, the carry save adder. You may also want to look up '4:2 compressor', which uses a similar principle.

An iterative multiplier can feature **early termination** by spotting that all remaining partial products are going to be zero.

Booth's Multipliers

[*This is, maybe a bit advanced, but may be of interest to some.*]

Booth's multiplication tries to reduce the number of partial products by multiplying by strings of consecutive '1's.

This works by turning a string of '1's into an addition and a decrement. For example:

- 111 can be rewritten as (1000 - 0001) (7 = 8 - 1)
- 11111 can be rewritten as (100000 - 000001) (31 = 32 - 1)

These can then be evaluated in two operations, regardless of the number of bits in the string.

Unfortunately, the irregularity of the processing works against a pure Booth's multiplier implementation. A 'modified Booth encoding' is more common.

Modified Booth encoding

For a two-bit at a time multiplication, divide the multiplier word as below:

```
Multiplier      1 0 1 1 0 1 0 0 1 1 1 0 | 0
Sub-multipliers ( 1 0 1   0 1 0   1 1 1 |
                    1 1 0   0 0 1   1 0 | 0
```

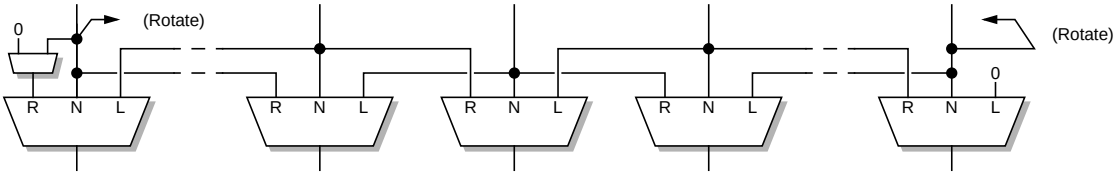
Each three-bit field represents two bits of the multiplier: the right hand bit represents a sort-of 'carry input' which can be 'assumed' from its less significant neighbour. Therefore, for example, a field '101' represents multiply by ((2+0)+1) = 3. However, as the top bit is 1 the more significant neighbour will assume a carry in (two bits up) with a significance of 4, so locally we multiply by -1. Thus, locally, the multiplications required are: {-2, -1, 0, +1, +2} – all of which can be done by shifting, ANDing and inverting.

```
0 0 0   ×0  =×(0+0)
0 0 1   ×1  =×(0+1)
0 1 0   ×1  =×(0+1)
0 1 1   ×2  =×(0+2)
1 0 0   ×2  =×(4-2)
1 0 1   ×3  =×(4-1)
1 1 0   ×3  =×(4-1)
1 1 1   ×4  =×(4-0)
```

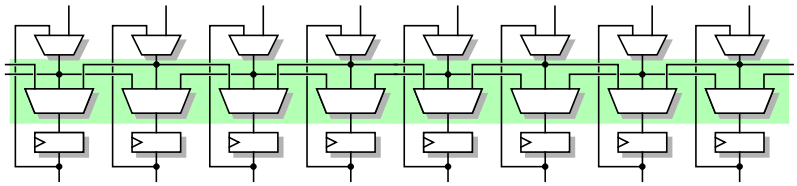
By multiplying two bits at a time the time taken can be halved (iterative) or the hardware requirements reduced (parallel).

Shifters

One-place shifter



The (bidirectional) shift register



Shifts multiple places at 1 place per cycle

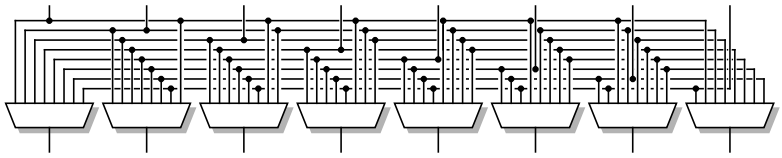
Based on structure above

Barrel Shifter

Shifts arbitrary number of places in one operation

Expensive in wiring

(Only 8-bit unit shown …)



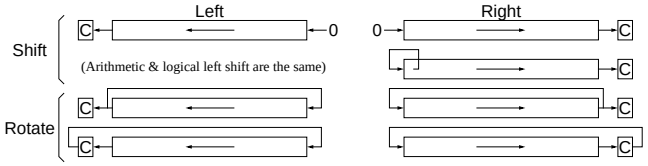
Shifters

Shifting is important in multiplication and division: it is also used for operations such as aligning and assembling bit fields into longer words. In principle, shifting is cheap to implement because it is simply wiring the bits to a different position. (All the bits move in the same way.)

- A left shift (one place) is a multiplication by two.
- A right shift (one place) is a(n integer) division by two.

Early microprocessors usually implemented only single place shifts, usually as an alternative ALU function. Multiplexers allow the selection of the particular shift.

A variety of shift operations are sometimes available:



A shifter can be extended to allow it to cycle, shifting a number of places. This is still quite cheap in hardware/wiring terms but will take a number of clock cycles to complete. (Doing this with a single instruction is still faster than writing a software loop, though.)

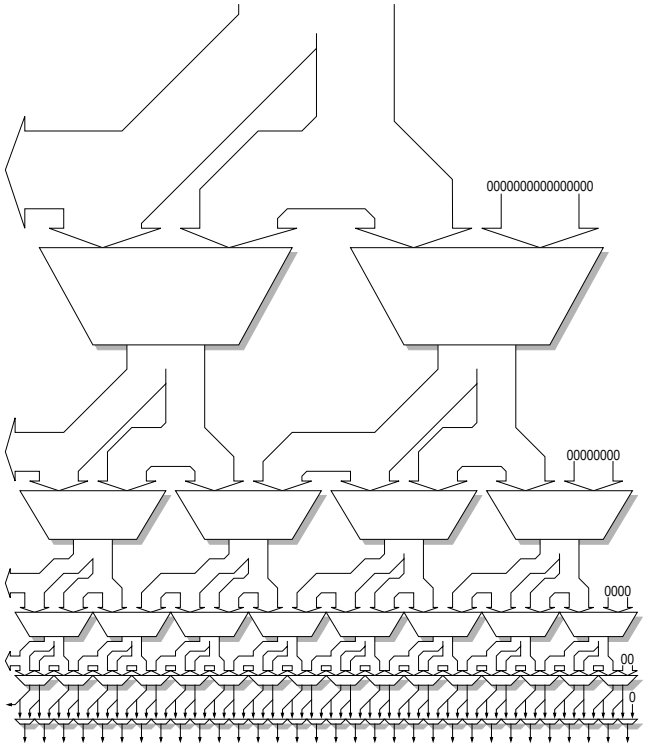
Barrel shift

A ‘barrel’ shift shifts/rotates an arbitrary number of places. This requires wider multiplexers and **a lot** of cross wiring (which occupies chip area). It may be able to operate in a single cycle, though.

The (8-bit) implementation shown on the slide hints at the complexity involved. Note that a barrel shifter only has to shift bits in one direction because (e.g.) on a 32-bit processor a right rotation of 8 bits is (almost[1]) the same as a left rotation of 24 bits.

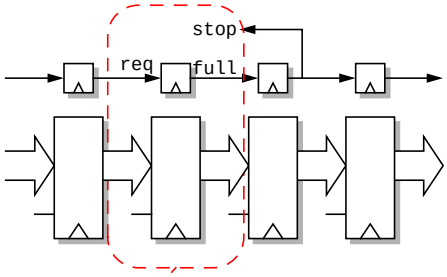
1. The carry output state may differ.

Funnel shifter

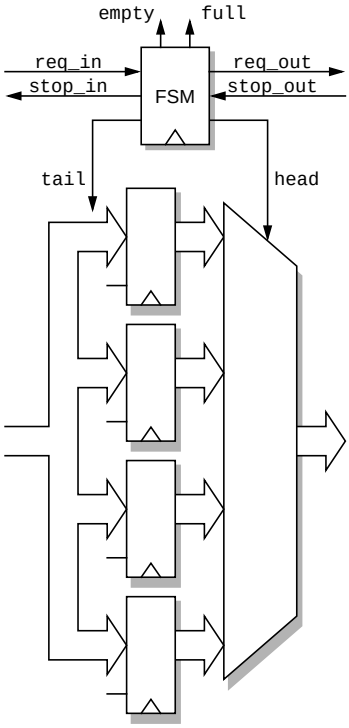


The figure shows a means of **barrel shift**ing using small (2:1) multiplexers: each stage shifts a set of bits by a power-of-two places (corresponding to one bit in the encoded shift distance). The figure shows a 32-bit left-shift only although it could be imagined that the ‘lost’ bits are returned instead of the ‘0’s on the right.

FIFOs



- ‘Circular’ buffer
- More complex (?)
- Low latency
- High throughput




```
always @ (posedge clk)
begin
if (!full && req) q <= d;
full <= ((!full && req) || (full && stop);
end
```

- Simple, functional, extensible
- Inefficient
  - High latency
  - Poor throughput

FIFO

A FIFO is a First-In First-Out buffer or a queue. Data words are inserted and will appear *in the same order* some time later.

Implementation

A FIFO can be pictured as a pipeline. Because this pipeline may stall, some control logic is required. A simple model (as shown on the slide) allows data to move when there is space for it in the next stage. This suffers from poor performance because, if data is packed into every stage only the word at the output can move. This is rather like a queue of traffic.

A more efficient model can look at the state of the whole FIFO. At a given point, data can move if the next stage is empty *or* it will become empty on the next cycle. To know the second condition requires applying the same condition recursively through the FIFO. This is a bit more complicated and, in particular, can require longer logic chains so may not be feasible in a very fast circuit.

Regardless, the minimum latency of either of these FIFOs is the number of stages in clock cycles: consider the transport of a word through an empty FIFO.

The other figure on the slide shows a ‘circular’ buffer. Space here precludes the full code this. Its basic operation is:

- There is a ‘head’ (output) and a ‘tail’ (input) address.
- The FIFO inputs a requesting element at the tail if it is not full (or if it is outputting a word in the same cycle); the tail address is incremented, modulo queue size.
- The FIFO offers for output the word at the head; if it is not empty and not ‘stopped’ it assumes the word has gone and increments the head pointer, modulo queue size.
- It starts empty with head and tail the same; it becomes full when it cannot accept more data without loss.

This form of FIFO can be good for energy economy because data is not *moved* except in and out. (For similar reasons it is a typical way of implementing a software FIFO.) It has low latency because, from empty, an input word becomes available for output on the next cycle. Throughput is one word per clock cycle.

Uses

FIFOs are used as buffers between functional blocks. They are useful when two interconnected blocks operate at similar rates *on average*, but their I/O rates vary over time. If an ‘upstream’ block wants to produce a burst of data it can be held in a FIFO rather than stalling the block; the ‘downstream’ block could then consume it (for example) more gradually.

The average ‘system’ throughput will still be limited by the slower block.

A practical FIFO has a finite capacity. It can become full and may have to stall the upstream block; it can become empty and may have to stall the downstream block. This comes under the (higher level) heading of **flow control**.

Hardware stacks

It is unusual to require stacks in hardware, but not unheard of. Their construction will typically be similar to a FIFO. Here are a couple of examples:

Procedure return

Most processors have powerful enough instructions to nest procedure calls in software and use the main memory to store the return addresses. Even with a simplified call mechanism (such as ‘BL’ in ARM) this is a reasonably complex operation; compare with 8080 (or x86) which pushes the return address externally, decrementing SP, before reloading PC with a target address.

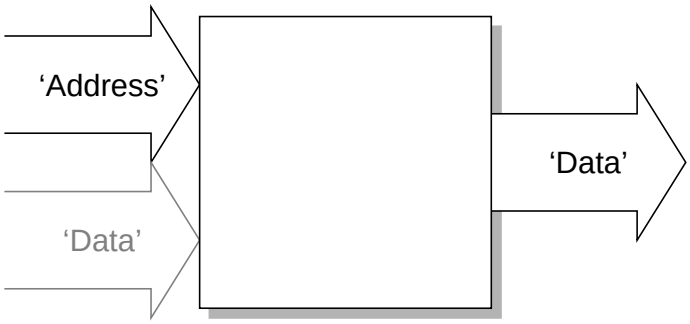
Some small microcontrollers (e.g. PIC) do not go to all this trouble and maintain a return stack in hardware to simplify (and speed up) the instruction execution. This is, necessarily, quite limited in capacity.

Nesting interrupts

With different interrupt priorities it may be desirable to let urgent, high-priority interrupts preempt lower priority service routines. *Something* has to keep track of the priority and restore priority levels correctly on returning. This is done with an **interrupt controller** (external device with e.g. ARM) which maintains a stack of the priorities which are currently in progress.

RAM

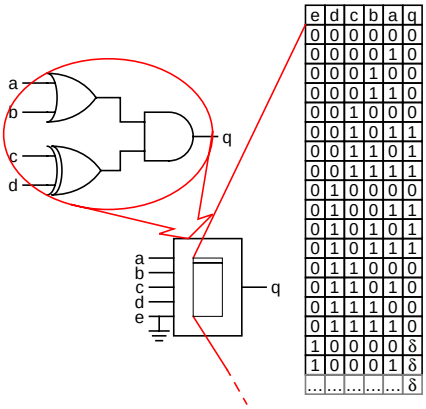
- RAM will be familiar as part of a computer architecture



Another use of RAM is as am embedded data store – e.g. for (large) buffers (FIFOs)

- Example storing a data packet from an Ethernet or USB interface

- In another sense a RAM can be a (writeable) look-up table
- Used for data 'look-up' tables in many structures
  - Routeing
  - Cache-type applications (BTB, TLB, …)
  - …
- Can also be used as a logic component
  - E.g. FPGA look-up



e	d	c	b	a	q
0	0	0	0	0	0
0	0	0	0	1	0
0	0	0	1	0	0
0	0	0	1	1	0
0	0	1	0	0	0
0	0	1	0	1	1
0	0	1	1	0	1
0	0	1	1	1	1
0	1	0	0	0	0
0	1	0	0	1	1
0	1	0	1	0	1
0	1	0	1	1	1
0	1	1	0	0	0
0	1	1	0	1	0
0	1	1	1	0	0
0	1	1	1	1	0
1	0	0	0	0	δ
1	0	0	0	1	δ
…	…	…	…	…	δ

RAM

Random Access Memory (RAM) should be a familiar component – at least as the 'thing that holds code and data for processors'. RAM is also used in numerous other applications. What actually constitutes a "RAM" may be slightly debateable: for example are an ARM's registers a RAM?

Register File

With a number of registers in a regular configuration, each register having a number (address) and being readable and writeable, a 'RAM' may be a reasonable description. For a processor such as ARM it is likely that the registers will be *implemented* as discrete instantiations when expressed in Verilog for a couple of reasons:

- A 'customised' RAM would be uneconomically small
- Register access is likely to be faster
- There will probably be several simultaneous reads and writes

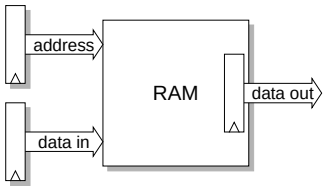
Nevertheless for some (larger?) register files using a true 'block of RAM' is a viable option.

RAM that is called "RAM"

The name "RAM" usually refers to a specific type of electronic structure which is optimised to store many bits in a small space. Each bit store is much more compact than using a flip-flop, although there is some overhead in the 'wrapper' which makes it uneconomic below a certain size. It is likely to be slower than flip-flops both to read and to write, although on-chip RAM access may fit within a single clock cycle. This will be Static RAM (SRAM): all this means is that its contents will be retained as long as it's powered.

A typical RAM will perform one action at a time: that is it can accept a single address and will either read a value from that address or write a (supplied) value to it as one operation.

Often, contemporary RAMs will have clocked registers on their inputs and outputs so they fit well with the synchronous FSM model. Thus to read a RAM an address is supplied at the beginning of a cycle and it provides the data output following the next active clock edge; writes follow a similar model.



A RAM's capacity is usually described by its:

- width – the number of bits in each word (single address)
- address length – N address bits can address 2^N words
- Total capacity – typically in *bits*

In ASICs RAMs are usually available as specialist blocks which can be synthesised to a size required by the user.

Because RAMs are generally useful, modern FPGAs typically contain blocks of RAM which will be used if possible and appropriate by the logic synthesizer. For this to be possible the design will need to employ the RAM in the 'correct' way, typically with synchronous I/O signals.

Multi-port RAMs

It's possible to make RAM with more than one 'port' so that multiple operations can be conducted (at different addresses) simultaneously. This can be useful but is expensive – significantly increasing the area/bit. In practice, is quite rare.

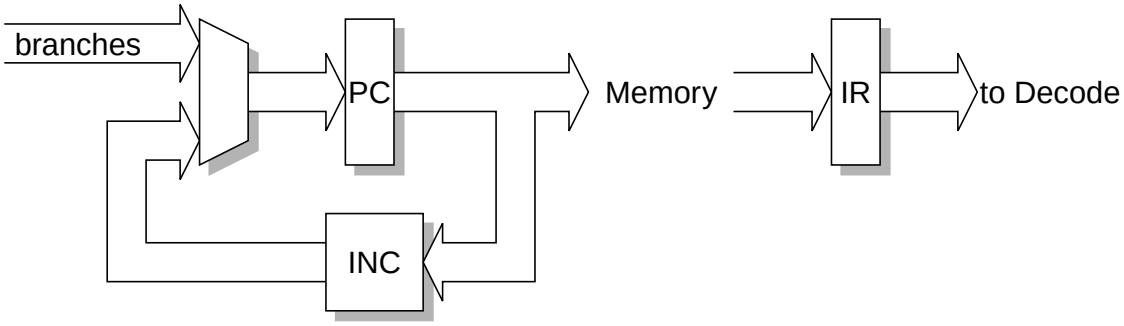
Some FPGAs allow multi-port operation to the extent of simultaneous read and write operations, with different addresses.

Logic

RAM can implement functions. Example: to add two 8-bit numbers would require a 64K ($2^{(8+8)}$) word RAM with 9 bit words (if the carry was wanted too). This approach has advantages (***any*** function in constant (short) time) but gets expensive as the word length increases. It is more economic for one input functions (e.g. trig. functions).

FPGAs use (small) RAM look-ups as basic, programmable logic elements.

Prefetching



- Separate PC from other architectural registers
  - ARM is slightly unusual in treating is as a general purpose register
- Include an extra incrementer
  - works in parallel with main ALU
  - PC is 'ahead' of the instruction being executed
- Needs provision for branches to load PC

Prefetching

- Prefetching tries to ensure that there is always an instruction ready to issue to the execution system by buffering words fetched from instruction memory.
- It is effective particularly when instructions vary in length (such as Thumb2 or x86 ISAs) to assemble a whole instruction in each cycle
  - Note: x86 instructions (for example) may be *longer* than the fetched word – although *on average* tend not to be.
- A prefetch buffer should be large enough to average out the differences in instruction length by never (rarely) becoming empty.
- Too big a prefetch buffer may fill up with the wrong instruction stream (by not noticing a branch) thus wasting time and power.
- Prefetch is a *speculative* process; the prefetch system 'guesses' what will be needed 'soon'.
  - The simple 'guess' is the next word in memory: right much of the time!
  - More sophisticated guesses use **branch prediction** to suppose that there was a *taken* branch at a certain address, and change flow (PC) accordingly.
  - Speculation can be wrong! In such a case the prefetch has to be discarded and corrections applied.
- The fetch width does not have to be a single word; wider buses can be used to get greater memory bandwidth.
  - Compilers may position branch targets on double/quad word boundaries to ensure a fetch cycle fills the prefetch buffer to the maximum extent; important when looping.
- The issue does not have to be a single instruction: **superscalar** processors will try to issue more than one instruction (to different functional units) at the same time.
- The issue does not have to be in order; instructions can be issued from a bigger 'pool' to avoid immediate *dependencies*. However the code still has to *look* as if it's in order to the software layer. (This can get involved!)

Branch Prediction

- A branch predictor's job is to try to ensure speculative fetches predict the flow the software will take on execution, keeping the prefetch buffer full(ish).
- The predictor does not know what an instruction is (it hasn't been decoded at the time!) so it must work with the PC and any history it has collected.
- A prefetch scheme is counterproductive unless it keeps the pipeline full (with useful stuff) most (nearly all) of the time. (Modern pipelines can be quite long.)

Very simple 'prediction'

- Fetch from the subsequent address of the previous fetch.
  - Works most of the time because most instructions don't branch.

Simple prediction

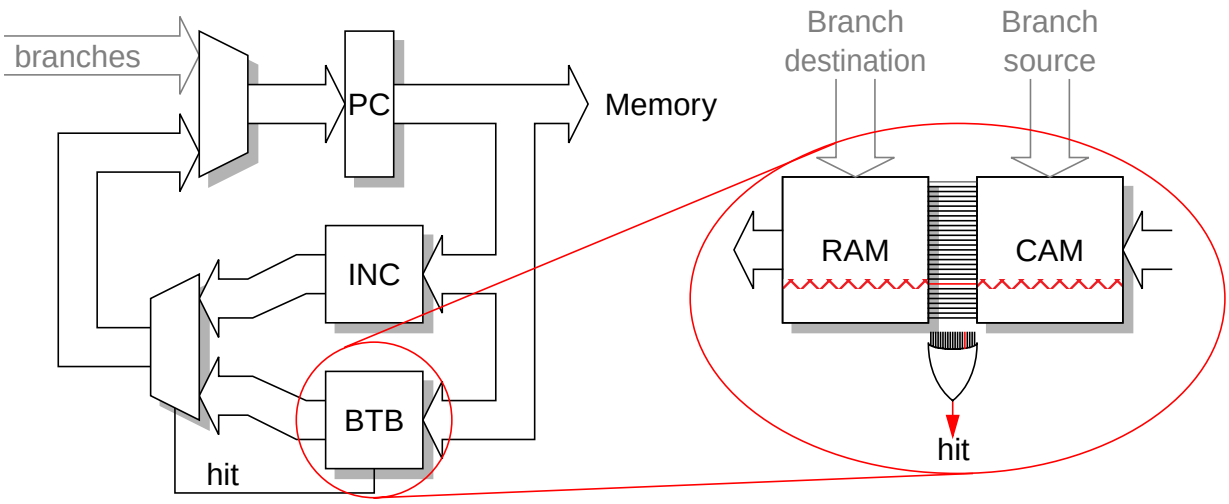
- Fetch from the subsequent address unless a branch has happened here before, in which case go to the target used last time
  - Works well because most branches are invariant
  - Doesn't take conditional branch behaviour into account
  - Not good for procedure returns &C.

More sophisticated prediction

- Keep some history of past branching behaviour and try to follow patterns.
- Note variant branches (such as returns) as they happen and relate them to 'calls' on a local stack

This is not a lecture on branch prediction algorithms – which are continually being refined – but it should illustrate some of the requirements and hint at the architecture.

Prefetch unit

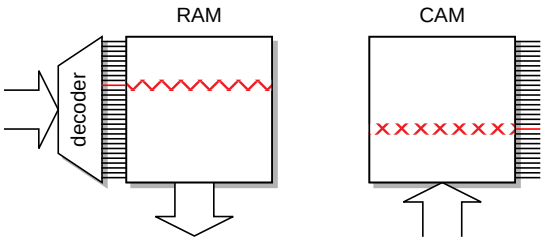


- An executed branch's details may be stored in the Branch Target Buffer (BTB)
- Subsequent PC values are associated with CAM contents
- If a match is found the (previous) destination may be substituted for incremented PC
- Other logic can improve decision making

Notes: BTB is in parallel with incrementer. BTB is a *cache*. Not all branches are cached.

Content Addressable Memory (CAM)

You should be familiar with the concept of RAM: present it with an address and out comes some data.



CAM works in the opposite way: present it with data and it outputs the address(es) where that data resides.

It is normal not to duplicate data in CAM so data resides in (at most) one place.

The names 'address' and 'data' are just *names* for buses.

The output may be binary encoded (not shown) or left as a 'one-hot' code which can be fed directly into an accompanying RAM

CAM is sometimes called '**associative memory**'.

CAM provides rapid searching because all the comparisons are done in parallel, thus a search takes a short, constant time (which may be a single clock cycle). There is, of course, a cost in power for this.

Incrementer

The incrementer is an adder but it can be simpler than a general purpose adder, adding '00…00' plus a 'carry' into the relevant bit position. Because one of the inputs is zero only half-adders are required at each position, although a carry can still propagate through the entire width of the word.

CAM structure

To be space- and power-efficient a memory is usually built from transistor-level circuits rather than gates.

A Static RAM (SRAM), as typically used on a CMOS logic chip, usually comprises six transistors per bit, arranged so they tessellate nicely in two dimensions. RAMs are common components; typically a RAM of the desired size will be synthesized by CAD tools.

A CAM needs to store data for comparison and comparison logic which amounts to an XOR per bit (XOR detects a *mismatch*) followed by an OR across the word (if any bit mismatches the whole word mismatches). This usually costs 9 or 10 transistors/bit, according to design style; with the wiring this makes it roughly twice the size of an SRAM of equivalent capacity.

CAM Uses

CAM provides parallel searches efficiently over a sparsely populated space. It is thus useful in some cache applications where a high degree of associativity is required. These include branch predictors, TLBs[1] and routers. [A router may use *tertiary* CAM where only a masked subset of the bits are compared.]

CAM is only rarely used in the 'ordinary' cache of a processor nowadays because of its high power requirement. Something like a 'level 2 cache' will typically be large enough to give good performance with low associativity.

In a BTB …

The CAM typically fails to recognise most addresses as the fetches occur. Occasionally it will spot an address which has previously caused a branch and it can then indicate the location of the corresponding destination in a RAM.

1. Translation Lookaside Buffers. Outside the scope of this module.