Computer Architecture

An embedded approach

Module 5

Faster and faster!

Contents

Speeding up

It makes sense that a higher speed clock would make a CPU run faster, but unfortunately we soon hit some limits. So other options are needed. Many have been tried:

Do more per clock cycle (CISC)

Do less per clock cycle (RISC)!

Perform operations in parallel

Speed up some common operations

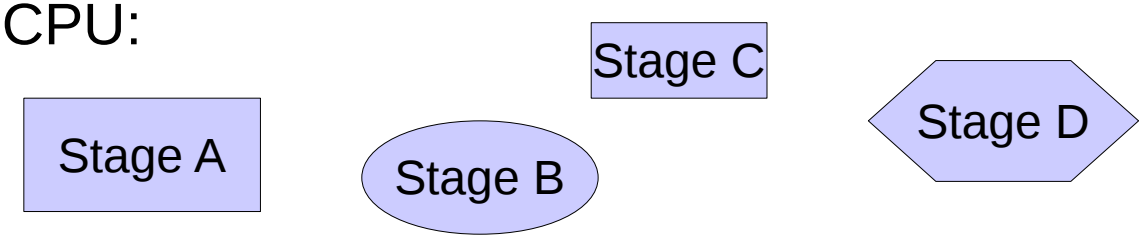
Speed up some complicated operations

Identify and fix performance bottlenecks

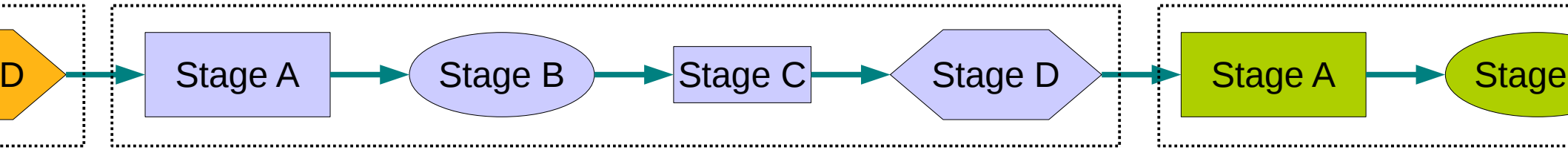
We're going to look at each of these approaches over the next few pages, starting with the most common response: pipelining.

Pipelining

Imagine you need to do four things (or “stages”) to process each instruction in a CPU:



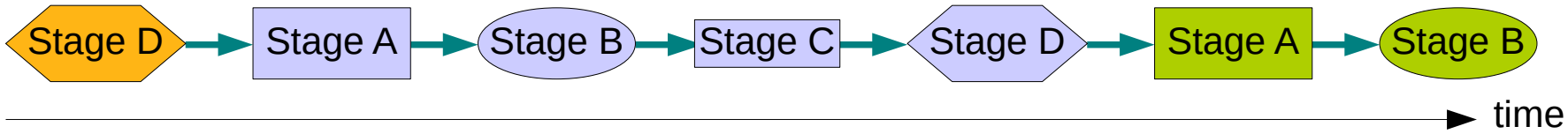
If we perform these in sequence, as fast as possible, the instructions are a long way apart, and a program will be slow:

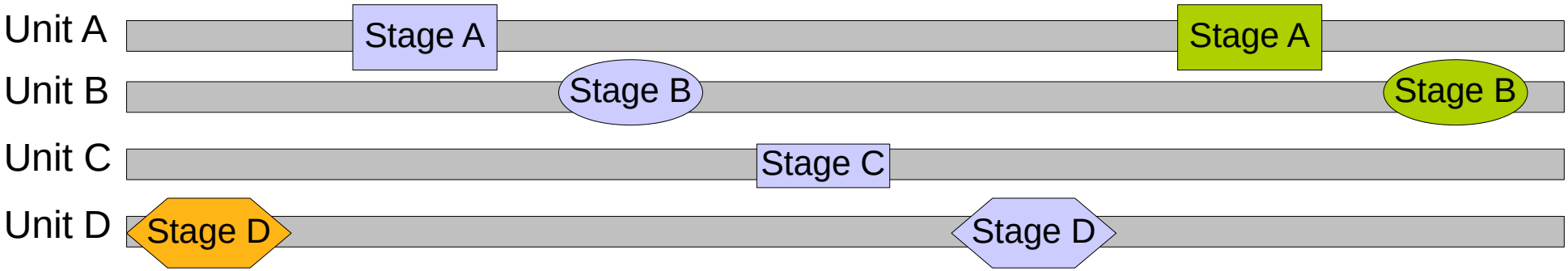


However, it is important to realise that because the stages are different, they actually make use of different hardware... so the hardware unit that performs Stage A is different from the unit that does Stage B, and so on.

Pipelining

Let us look at what the hardware units are doing:

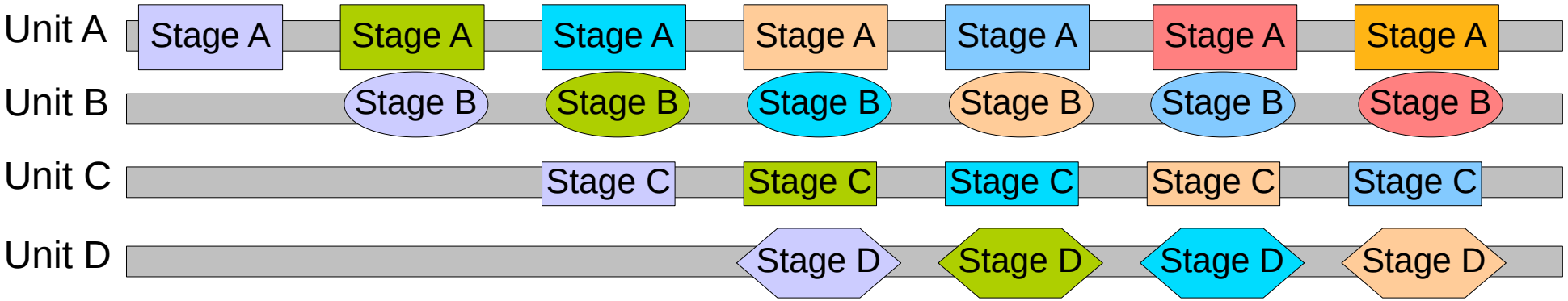




None of the hardware units looks busy... most of them are only doing something for a short period of time, then are idle.

Pipelining

Can we make the hardware units work harder? Yes:



what we have done is allowed each instruction to overlap the previous one.

While Unit A is handling Stage A for one instruction, Unit B is handling Stage B for the previous instruction...

Pipelining

In real pipelined CPUs, stages might include:

Fetch instruction
Decode instruction
Fetch operand
Execute instruction

For now, it does not matter exactly which stages are present, because we are only exploring the concepts (and every CPU does things slightly differently), but the sequence of operations is important, and the fact that these operations can overlap – ***SOMETIMES!***

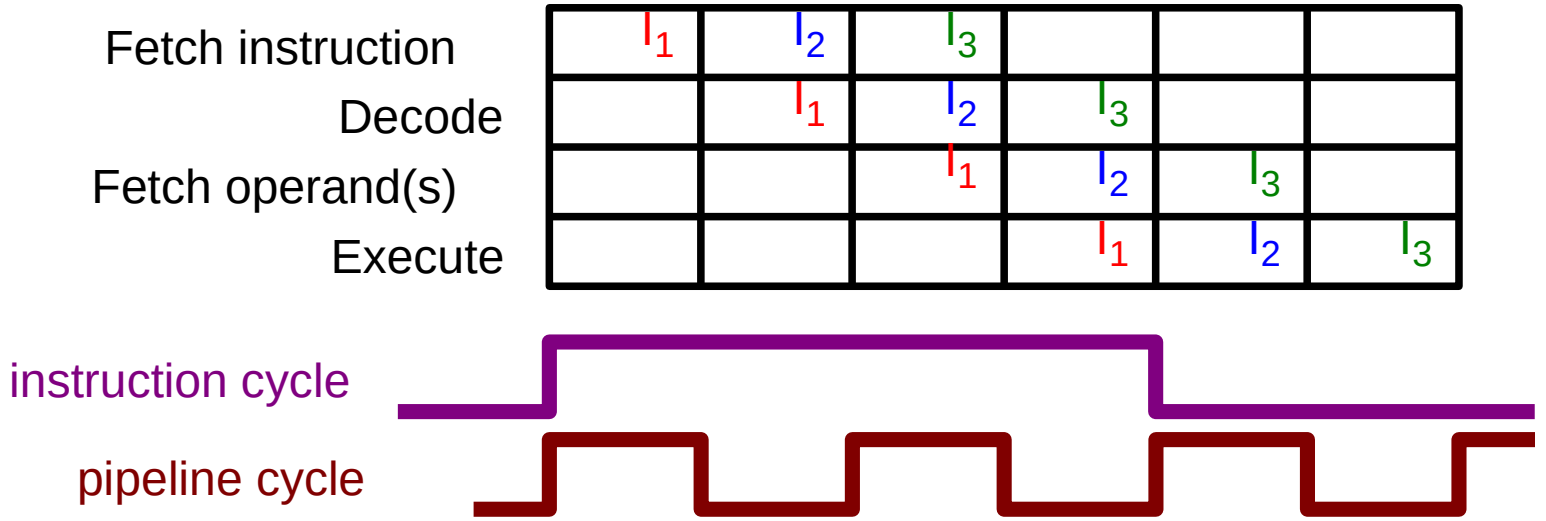
Pipelining

To explore this, consider what happens when we execute instructions. There are several phases to execution (assume an ARM style CPU):

FETCH	Fetch instruction from memory to the instruction decoder
DECODE	Decode the instruction and decide what to do with it
LOAD	Load the operands for the instruction
EXECUTE	Send the instruction to the correct execution unit for processing
RESULT	Collect the results of calculation for updating status flags
SAVE	Save the result in the correct location

Pipelining

Invented by IBM, a pipeline is a process that often increases the length of time that a CPU takes to process an instruction, but allows the instruction operations to overlap, so increasing throughput.

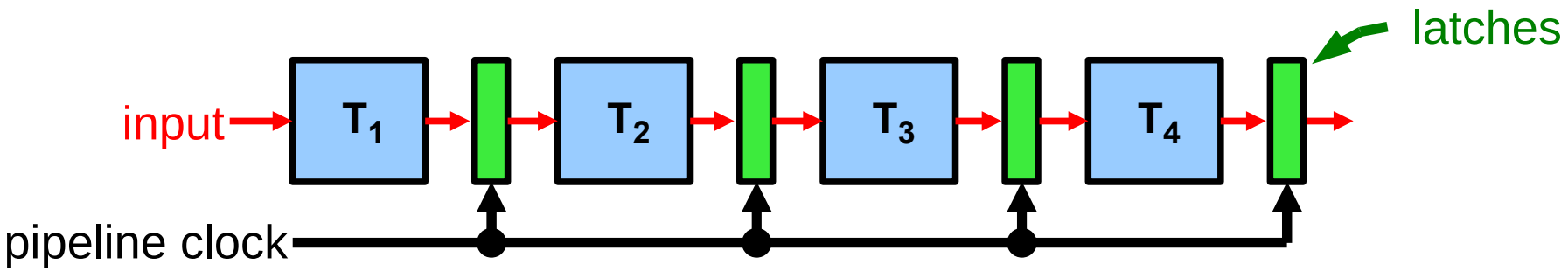


In the example above, 4 instructions finish in one instruction cycle. Without a pipline, one instruction must finish before the next one starts...

Pipelining

So an *n*-stage pipeline means that the CPU can operate at a maximum of *n* times faster than without a pipeline.
The picoJavaII has a 6-stage pipeline, whilst the ARM7 has a 3-stage pipeline. The ADSP2181 has no hardware pipelining.

To pipeline any task, it is necessary to split the task into a number of sequential subtasks, and connect these linearly in a serial fashion:



Synchronisation is maintained with latches.

Pipelining

Commonly, pipelines combine some of these functions into single units. There are many possible designs of pipeline, but here are some you can find in modern processors:

3 stage (F – E – S)
FETCH & DECODE – LOAD & EXECUTE & RESULT – SAVE

4 stage (F – D – E – S)
FETCH – DECODE & LOAD – EXECUTE & RESULT – SAVE

5 stage (F – D – L – E – S)
FETCH – DECODE – LOAD – EXECUTE & RESULT – SAVE

Normally the slowest processes live in a unit on their own.
These are often going to be the ones that do:
1. Memory accesses (load or save)
2. Some sort of complicated calculation (FPU?)

Pipelining

RISC processor pipelines have grown over the years:

ARM2 (1988)	3 stages	**FETCH – DECODE – EXECUTE**
ARM9 (1998)	5 stages	**FETCH – DECODE – EXECUTE – MEMORY – WRITE**
ARM11 (2002+)	8 stages	**FETCH1 – FETCH2 – DECODE – *EX* – WRITE**
		***EX* for ALU is: SHIFT – ALU – SATURATE**
		***EX* for MAC is: MAC1 – MAC2 – MAC2**
		***EX* for MEM is: ADD – CACHE – WRITE**

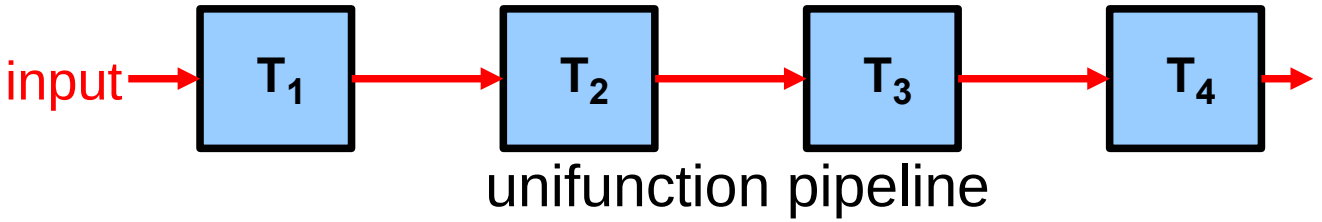
Some research machines have even longer pipelines.

Get some background about the ARM (but not the pipeline):
http://tisu.it.jyu.fi/embedded/TIE345/luentokalvot/Embedded_3_ARM.pdf

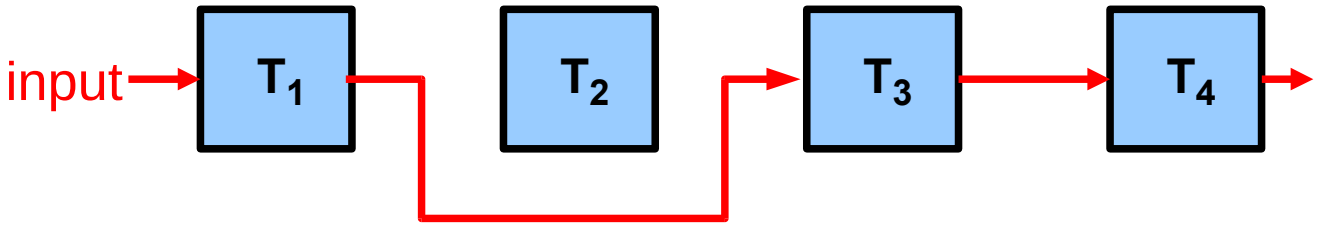
If you are having trouble understanding the concept of a pipeline, look here:
http://www.pcmech.com/article/understanding-processor-pipelining

Pipelining

Pipelines can be uni or multi-function. The latter allows different routes to be taken through the pipeline, but is very diff cult to control in hardware.



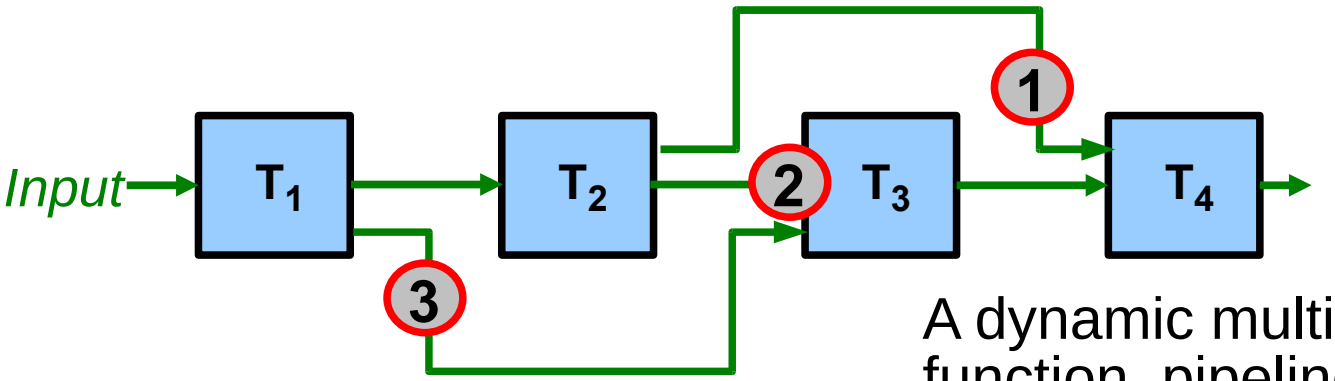
unifunction pipeline



multifunction pipeline - set up
to perform a different function

Pipelining

A multi-function pipeline can change it's conf guration either dynamically, in which case it allows a different route to be taken by every instruction, or statically, in which case it must be cleared of instructions before the alteration occurs.



A dynamic multi-function pipeline

1 2 *f rst and second instructions*

possible instruction paths

Pipelining

The slowest instruction in a pipeline will be the bottleneck because all tasks are clocked with equal duration.

Given a program with *s* instructions. Each instruction needs *n* clock periods to execute. The total time required to run this program with no pipeline is $T_I = sn$

With an *n*-stage pipeline and 1 clock period between stages the total time required for execution is $T_N = n + (s - 1)$

(it takes n clock periods for the first instruction to run through the pipeline, plus s-1 periods until all the other instructions have finished)

Speedup is defined as $S_N = T_I/T_N = sn/(n + s - 1)$

and as s -> , S_N -> n

(potential maximum speedup with infinite instructions)

Pipelining

A measurement of pipeline eff ciency takes into account that some units are not being used during start-up and end;

Eff ciency= aggregate unit operating time/(n × pipeline operating time)

= s / (n + s -1) [actual speedup / theoretical maximum speedup]

but look at this equation - *it's almost identical to the speedup equation on the previous page!* So we can say that;

Eff ciency= S_N/n

Throughput is the number of tasks completed per unit time;

throughput = s / (n + s -1) = eff ciency !

Pipelining

Pipelines work well when they are continuously operating and full. Eff ciency drops if they empty (either deliberately in order to change a static pipeline, or because of the sequence of instructions).

A sequence of NOPs could empty a pipeline....

And so could a conditional branch:

I_1	ADD	R0, R0, R1
I_2	AND	R4, R2
I_3	SUBS	R2, R0
I_4	BGT	loop

In this example code segment, the branch depends on the previous instruction. This needs to run completely through the pipeline before it is known whether the branch is going to be taken, so the pipeline stalls to wait for that instruction.

Pipelining

One way to minimise the pipeline stall is to continue executing the instructions following the branch speculatively. If the branch does not occur then there was no pipeline stall. If, however, the branch does occur, the pipeline must be flushed of the speculative instructions. On average, this reduces pipeline stalls by 50%

Some CPUs have multiple pipelines (IBM 370). One pipeline assumes the branch is taken, the other assumes it is not. The incorrect pipeline is flushed, and there is no penalty (*but this assumes only two branch alternatives*). Some advanced CPUs predict branches based on past history (either local for this branch, or global). See later!

Finally, processors like the TMS320C5XX provide delayed branch instructions: forcing the programmer/compiler to deal with the issue.

Pipelining

Often, a good compiler will re-order instructions to optimise their performance when using a pipelined processor, and to reduce branch penalties. Some compilers insert NOP commands to deal with delayed branches. This can sometimes improve performance.

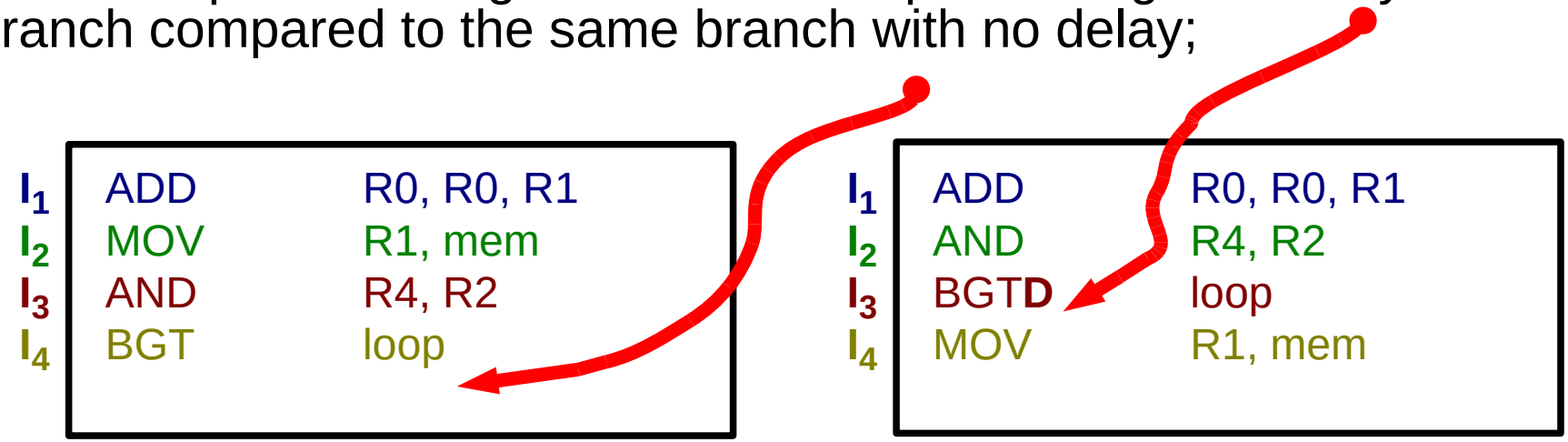
Multiple conditional branches within a short area of memory can compound branch penalties, and only those CPUs with delayed branch instructions can processes computed jumps without loss of eff ciency.

Unconditional branches cause less problems: the pipeline simply follows the branch.

Pipelining

Delayed branches:

The example code fragment shows the positioning of a delayed branch compared to the same branch with no delay;



The B*cc*D instruction goes into the pipeline and the branch made after the next instruction ends (i.e. when the outcome of I_2, which determines the branch, is known). So the pipeline is not stalled!

But with the B*cc*, the pipeline must wait until the previous instruction completes. Only then will it know if it should take the branch or not.

Pipelining

Pipeline hazards: data dependency

RAW (read-after-write) hazards arise when one instruction relies upon the output from the previous instruction:

```
ADD     R0, R2, R1     ;R0=R2 + R1
AND     R1, R0, #2;R1=R0 & 2
```

In a pipeline, there would have to be a stall, waiting for the ADD to complete before the AND commenced.

There is also an antidependency or WAR (write-after-read) hazard here, because the AND must not change the value of R1 before the ADD has used the value that is in R1.

Pipelining

Pipeline hazards: data dependency

WAW (write-after-write) hazards arise when two nearby instructions must write to the same location, but that location is needed by a third instruction:

```
ADD     R0, R2, R1    ;R0=R2 + R1
AND     R1, R0, #2    ;R1=R0 & 2
SUB     R0, R3, #1    ;R0=R3 - 1
```

The AND has a very short time window when it can access R0. If it reads too early or too late, it gets the wrong value.

In non-pipelined, or simple static pipelined machines there is no problem, but imagine the diff culties for multi-pipeline machines or dynamic pipelined machines.

Pipelining

Pipeline hazards: detection and solution

A good compiler should detect these hazards, and reschedule instructions accordingly (or insert NOPs to separate instructions).

In summary, dependencies between instructions ***i*** and ***j*** exist if there is;
anything that ***i*** can do to affect the result of ***j*** or
anything that ***j*** can do to affect the result of ***i***

Including:
The same registers being used for input or output, the same functional unit(s) being needed, some mode change or condition setting by one of the instruction when the other instruction is conditional.

Pipelining

Pipeline hazards: detection and solution

Processors should check for hazards and resolve them either by introducing a pipeline stall, or by a hardware technique such as data forwarding.

This is the technique of passing the result of one function directly to another function, without passing through an intermediate register first. This can often be done in software as well as through hardware. A software example (done by compiler) could be:

```
MOV   $1000, R0          replaced by:      MOV   $1000, R0
MOV   R1, $1000                            AND   R0, R0, R2
AND   R0, R1, R2
```

This is called store-fetch forwarding

Pipelining

Pipeline hazards: detection and solution

Fetch-fetch forwarding is slightly different, multiple reads from a single memory location are replaced by holding that value in a register (or even in a fast on-chip stack):

```
MOV   R0, $1000          replaced by:     MOV   R0, $1000
ADD   R2, R0, R1                          ADD   R2, R0, R1
MOV   R1, $1000                           ADD   R3, R2, R0
ADD   R3, R2, R1
```

Store-store forwarding replaces multiple writes to one memory location by intermediate storage in an internal register, followed by a single, final write.

Pipelining

Pipeline hazards: detection and solution

These methods of fetch-fetch, store-store and store-fetch data forwarding are usually optimisations performed during compilation by a good compiler. However similar operations can occur in hardware directed at run-time by the CPU.

```
ADD   R2, R0, R1
SUB   R2, R2, R0
```

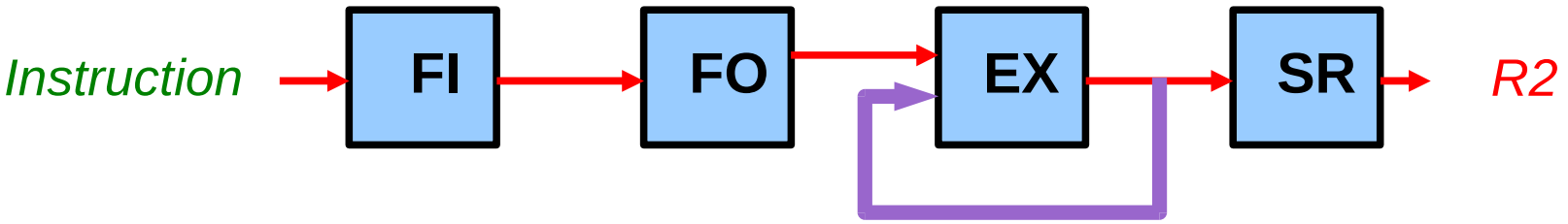
In this code fragment, the SUB instruction requires the output from the ADD as an operand. This is passed in register R2.
It also writes its own output to the same register.
This demonstrates more than one type of data dependency.

Pipelining

Pipeline hazards: detection and solution

Consider the pipeline path for this code fragment:

```
ADD   R2, R0, R1
SUB   R2, R2, R0
```



forward the result of the previous instruction (R2)

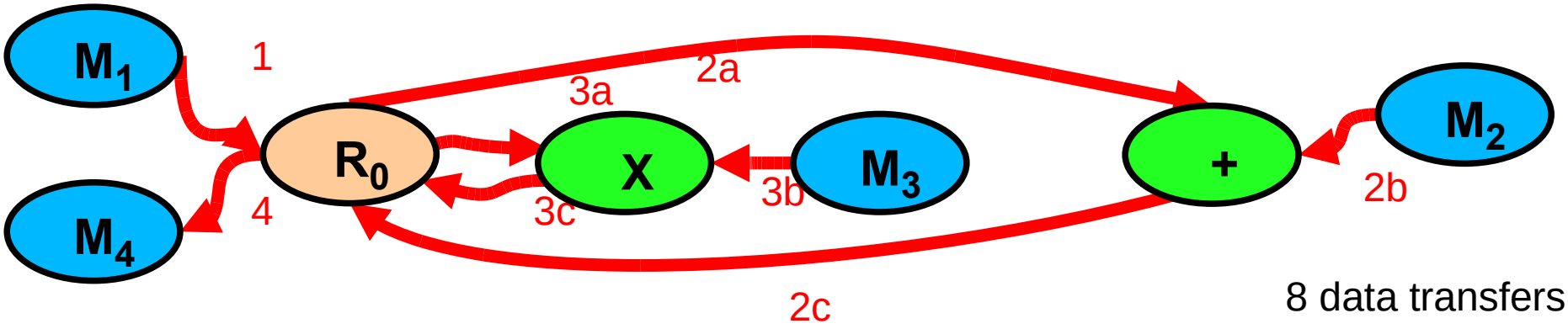
The value that is supposed to store to R2 after the ADD is passed directly as one of the SUB instruction operands. The first store to R2 is skipped - the value will be overwritten by the second store anyway.

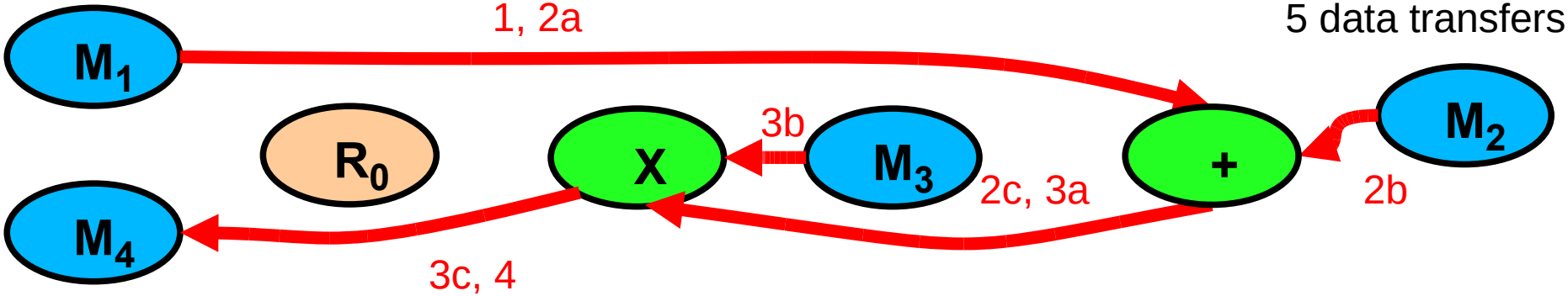
FI: fetch instruction, **FO**: fetch operand, **EX**: exectute, **SR**: store result

Pipelining

Another example of data forwarding:

```
MOV     R0, m1      (1) R0=m1
ADD     R0, R0, m2  (2) R0=R0 + m2
MUL     R0, R0, m3  (3) R0=R0 x m3
MOV     m4, R0      (4) m4=R0
```





RISC/CISC and Microcode?

How does a pipeline know what stages the current instruction needs to go through? Or which processing unit to send operands to? Or whether to stall the pipeline at times?

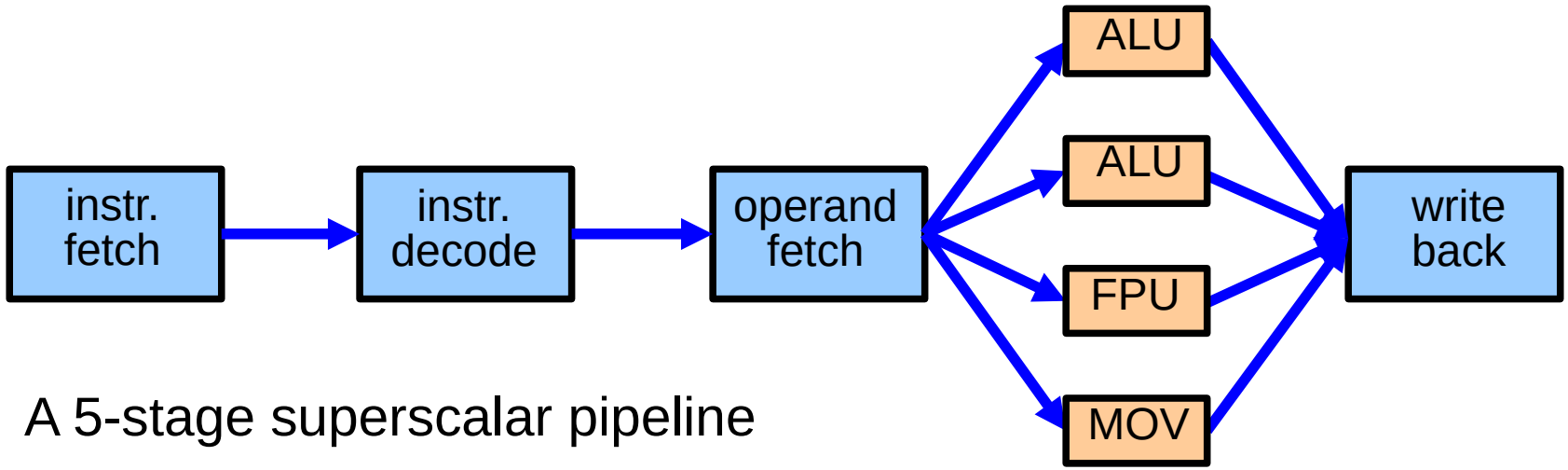
In a simple CPU, this can be directed by a hardwired control bus, but in a CISC machine, or a machine with a complex pipeline, the control hardware could become more complex than the CPU....

In the previous module we met one control solution: microcode. The technique was invented at Cambridge University in the 1950s but is still used today on some of the modern CISC processors.

RISC machines usually have simple, regular, single-cycle instructions and a load-store architecture.
All attributes that make pipeline design easy!

Superscalar

As pipelines become more complex, the opportunity for hazards increases. Multi-pipeline machines, in particular, require considerable resources devoted to detecting and preventing hazards. One solution is to keep a single, linear pipeline, but provide it with multiple functional units. This is called a superscalar architecture.



A 5-stage superscalar pipeline

Superscalar

With the superscalar pipeline just presented, the operand fetch unit can actually issue instructions at a faster rate than any of the processing elements can handle. It makes sense to interleave instructions to the two ALUs, the FPU and the load/store unit (MOV).

The most time consuming element (the FPU) does not need to hold up the pipeline while it is operating because other non-dependent ALU and memory move jobs can still be handled.

The Pentium II has this type of superscalar architecture whilst earlier pentiums had dual pipelines (and the 486 had only a single pipeline).

Superscalar

Often the different execution units will run at different speeds. Here is an example:

fetch & decode	*ADD*	*SUB*	*AND1*	*FADD*	*NOT*	*MUL1*	*MUL2*	-	-	-	*NOR*	*AND2*	*NOT*
ALU		*ADD*		*AND1*		*NOT*							
ALU			*SUB*										*NOR*
FPU					*FADD*								
MUL							*MUL1*				*MUL2*		
store result				*ADD*	*SUB*	*AND1*		*FADD*	*NOT*		*MUL1*		
Clock cycles:	0	1	2	3	4	5	6	7	8	9	10	11	12

This machine has 2 ALUs, 1 FPU and 1 MUL.

- When two instructions finish execution simultaneously, the earliest issued gets stored first.
- In 12 clock cycles only 6 instructions finish...

Superscalar

Interestingly, we can substantially improve the machine of the previous page if we place a one instruction buffer after the fetch & decode unit:

fetch & decode	*ADD*	*SUB*	*AND1*	*FADD*	*NOT*	*MUL1*	*MUL2*	*NOR*	*AND2*	*NOT*	*NOP*		
buffer								*MUL2*					
ALU		*ADD*		*AND1*		*NOT*				*AND2*			
ALU			*SUB*						*NOR*		*NOT*		
FPU					*FADD*								
MUL							*MUL1*				*MUL2*		
store result				*ADD*	*SUB*	*AND1*		*FADD*	*NOT*		*MUL1*	*NOR*	*AND2*
Clock cycles:	0	1	2	3	4	5	6	7	8	9	10	11	12

Now 8 instructions complete in 12 clock cycles. The buffer just stores instructions that can't execute yet because the functional unit that they require is not free...

Note: this doesn't fix issues of dependency (i.e. if NOR input requires the result from MUL2 then we can't do something like this) – see later.

Instructions-per-Cycle

Instructions-per-Cycle (IPC) measures how quickly (in clock cycles) a program of given length gets executed.

CISC: IPC usually << 1.0
but some instructions can do a lot!

RISC: IPC approaches 1.0
simple, short instructions that execute quickly

Superscalar: IPC approaches *n* (for *n* issues per cycle)
usually heavily pipelined RISC-based designs

VLIW/EPIC: IPC >> 1.0
(see module 9)

Parallel: *n* cores approach *n* times their individual IPC
but lots of questions remain - can we achieve this?

Hardware acceleration

Hardware acceleration can include co-processors dedicated to performing specialised tasks (FPU or MMX), or other dedicated hardware speed-ups such as:

- zero-overhead loops
- dedicated address generation units
- shadow register sets
- hardware stack
- inter-processor communications

Hardware acceleration

Zero-overhead loops:

Many DSP processors especially need to execute code that consists of compact, fast, loops. These could be *for(), while(),* or *do()* loops.

A standard CPU needs to set up a register with the control variable, add/subtract from it at the start or end of the loop, and perform a comparison for end-of-loop condition.

The instruction cycles needed to perform these operations are termed the loop overhead.

Hardware acceleration

The loop overheads can be reduced in two main ways.

1. address generation hardware (*see later*)
   a separate hardware unit contains registers and a simple ALU which can load a loop variable and its step, and terminate the loop on fulfilment of a simple condition

2. a PC trap
   a register is loaded with loop end-value. When the PC is found to hold this address, its increment to next instruction is cancelled, and it is reset to the start of the loop. The loop condition is checked at the top of the loop.

Hardware acceleration

Zero-overhead loops:

Standard loop:

```
MOV       R0, #count
loop      .....
          .....
          (body of loop)
          .....
          .....
          SUB       R0, R0, #1
          BGT       loop
(outside the loop)
```

Zero-overhead loop (ADSP2181)

```
MOV       CNTR, #count
DO loop UNTIL LE
          .....
          .....
          (body of loop)
          .....
          .....
loop (outside the loop)
```

Hardware acceleration

Zero-overhead repeat (TMS320C50)

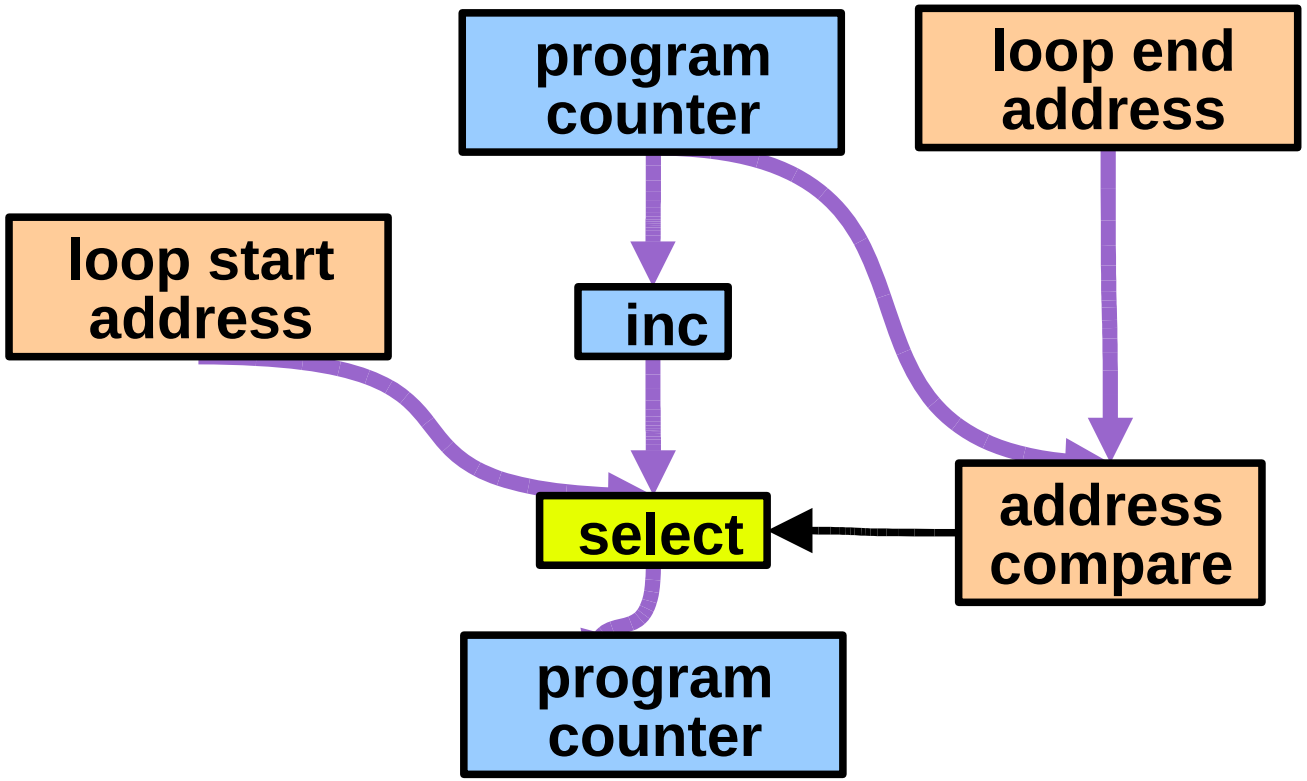
```
MOV   BRCR, #count
RPTB  loop-1
      .....
      .....
      (body of loop)
      .....
      .....
loop  (outside the loop)
```

For a loop repeating 512 times, a standard CPU would have an overhead of 1 instruction outside the loop, and 2 instructions inside totalling 1025 wasted cycles.

A zero-overhead loop may have one or two set-up instructions followed by no overhead inside the loop: a total of only 2 wasted cycles.

Hardware acceleration

Zero-overhead loops: how to design the hardware?



Hardware acceleration

Dedicated address generation units are commonly used to perform addressing arithmetic in parallel with data processing operations. This is most successfully used in the ADSP2181 which has two data address generators (DAGs) and a set of dedicated address registers.

The alternative is to take the approach of the ARM and perform all arithmetic (data and addressing) with a single ALU and a bank of general purpose registers.

We now investigate both approaches...

Hardware acceleration

Dedicated address generation units:

The ADSP2181 DAGs each contain four I, L and M registers;

I0	L0	M0
I1	L1	M1
I2	L2	M2
I3	L3	M3

DAG1

I4	L4	M4
I5	L5	M5
I6	L6	M6
I7	L7	M7

DAG2

The I (index) registers contain actual addresses whereas the L register contains a length and the M registers are address modif ers.

There are different ways of using the DAGs:

Hardware acceleration

Dedicated address generation units:

For address register indirect addressing, an operand is fetched from the address in the I register (of course, it can be used directly too).

Address register indirect with post increment/decrement can be used by specifying an M register in the instruction. Any M register in the same DAG as the I register can be used to modify the I register after the access (the M register is *signed* so it can hold a negative value).

The L registers are used to implement wrap-around or circular addressing. If an L register corresponding to any I register is non zero, then when the I register has been modif ed (by any M registers) so that the I register has reached an offset from its original value equal to the L register, the I register is set back to its original value.

Hardware acceleration

Dedicated address generation units:

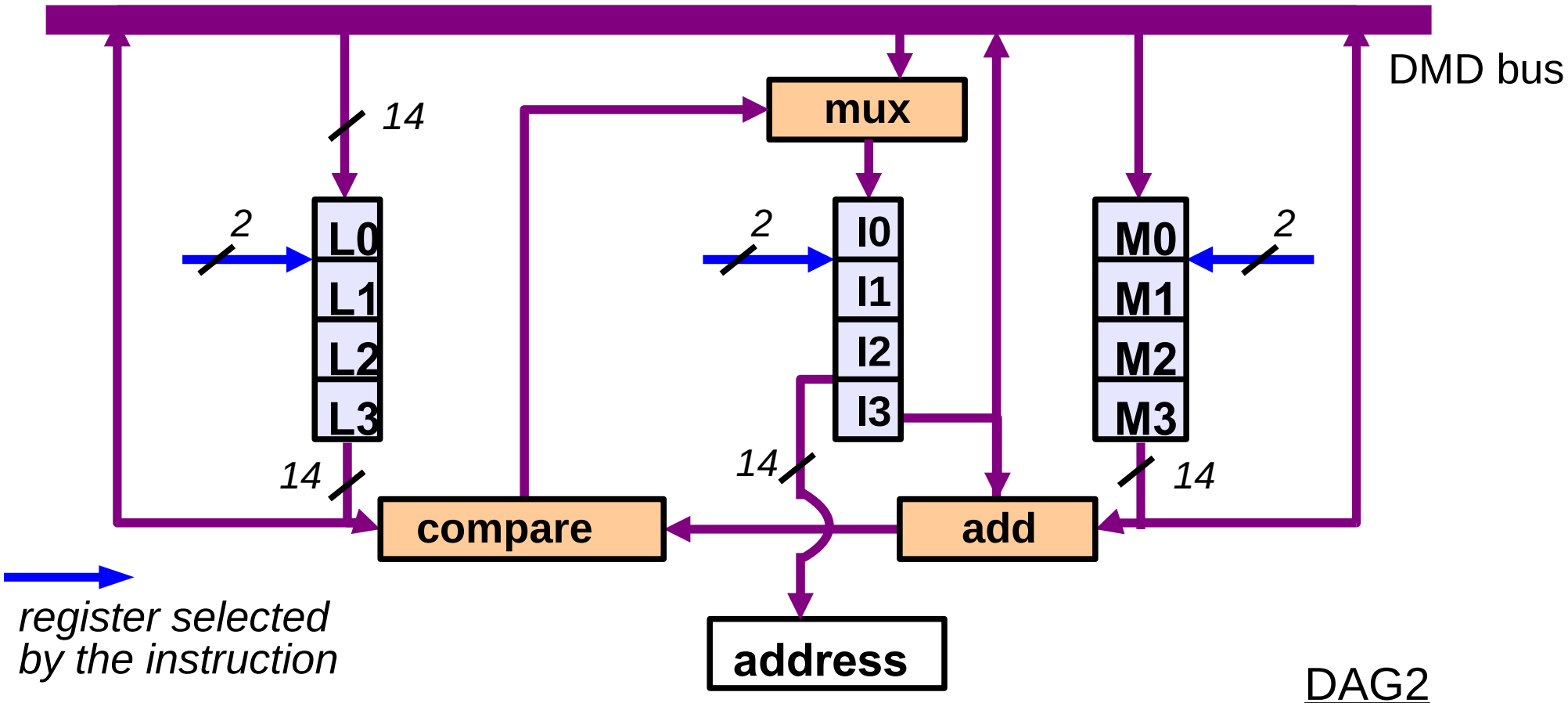
There is actually one dedicated ALU for each DAG. This can add any of the 4 M registers to any of the 4 I registers, compare the new I value with the L register, and store the new I value (either the added value, or the reset value) back to the register.
This all happens within a single instruction cycle.

Having two independent DAGs means that both operands to an instruction can use any addressing format, and still complete in a single instruction cycle.

In addition, one of the DAGs can bit reverse its addresses (useful for doing an FFT, Fast Fourier Transform).

Hardware acceleration

Dedicated address generation units: of the ADSP2181



DAG2

Hardware acceleration

Dedicated address generation units: of the ADSP2181

ADSP2181 assembly language has special instructions to help manage the complexities of the dual DAGs:

```
reg=DM(I3,M2)      ; loads reg with the value at data memory
                   ; location I3, then adds M2 to I3.  If L3=0
                   ; then I3 is set to the new value (otherwise...)
```

How to set up and use a circular buffer:

```
L0=128             ; buffer is 128 memory locations long
I0=buffer          ; I0 holds address of the start of the buffer
M0=2               ; inc. buffer by 2
M1=-1              ; or by -1 each time it is accessed
```

Hardware acceleration

Dedicated address generation units: of the ADSP2181

```
AX0=DM(I0,M0)      ; read buffer element 0, pointer set to 2
AY0=DM(I0,M1)      ; read buffer element 2, pointer set to 1
```

If this instruction is placed in a loop, it will read each of the elements in the circular buffer into AX0 and AY0 in turn:
First time around,
AX0=buffer[**0**] and AY0=buffer[**2**]
second time around,
AX0=buffer[**1**] and AY0=buffer[**3**]
....
eventually,
AX0=buffer[**126**] and AY0=buffer[**0**]
and then,
AX0=buffer[**127**] and AY0=buffer[**1**]

Hardware acceleration

Dedicated address generation units: of the ADSP2181

The on-chip data and program memory of the ADSP2181 and the powerful dual Data Address Generators, this CPU can access operands in two memory locations indirectly with post-increment and wraparound, as well as process the instruction that uses those operands, *in a single instruction cycle*.

Hardware acceleration

Dedicated address generation: of the ARM7

By contrast, the ARM chip, being a RISC design seeks to minimise specialised hardware (like DAGs), but to process every instruction as fast and as eff ciently as possible.

The ARM only has one instruction* to load data from memory to the register bank, and one to save it. This is called a **load/store architecture** (an extension allows multiple registers to be loaded or saved from sequential locations). Once data is loaded to a register, all internal operations can be performed on it.

The address to load from/store to can be indexed, with pre- or post-offset (increment/decrement). It can be direct or indirect.

* another *swap* instruction can do this but is designed for multiprocessor systems.

Hardware acceleration

Dedicated address generation: of the ARM7

Since the ARM does not have dedicated address ALUs, it must use the main ALU and associated functions. This has one major advantage (apart from reducing transistor count) in that it is more flexible: it is not confined to the DAG operations of the ADSP2181.

The ARM can use a special scaled register address indexing mode, where the address offset register can be multiplied or divided by a power of 2 before or after (pre- or post-) access:

LDR R0, [R1, R2, LSL#2]
This loads the value at location (R1+(R2*4)) into R0
(LSL means *logical shift left*)

Hardware acceleration

Shadow Register Sets

When implementing *re-entrant* code, it is necessary to store the context of the current registers at the beginning of the routine, and then restore them at the end. The state of the CPU at the end of the routine should be the same as it was at the beginning.

One example of when this is necessary is in interrupts because these can occur at any unpredictable time during program execution.

Consider an example of a serial port interrupt.

Hardware acceleration

For a PC-style serial port, a single chip usually interfaces to the CPU and outputs data serially. The chip has a built-in buffer of usually 16 to 64 bytes. It outputs data, emptying this buffer normally without the intervention of the CPU.
However, when the buffer is nearly empty, the serial chip *interrupts* the CPU to request that the buffer is refilled.

Since the buffer is still being emptied, the CPU needs to quickly fill up the buffer before it empties (which would cause an *underrun*). So an interrupt service routine (ISR) is used to service the interrupt and do this.

An interrupt can occur at any time: It is usually not possible to predict what the CPU will be doing at that instant. Most CPUs will finish the instruction they are currently executing, flush their pipelines and jump to the interrupt service routine (via an interrupt vector).

Hardware acceleration

A serial port ISR would need to do the following:

1. *save the state of the CPU*
2. *disable other interrupts (possibly)*
3. *get data from memory and pass it to the serial chip*
4. *restore the state of the CPU*
5. *re-enable interrupts*
6. *jump back to the code it was executing before interrupt*

The state of the CPU includes all status registers, and all other registers that might be changed by the ISR. Registers that are not going to be changed by the ISR do not need to be saved.
In an ARM this might be R0, R1, R2, R3, CPSR (status register).

If each write to main memory requires 4 instruction cycles, this totals 20 cycles (probably a lot more for a write-through cache).

Hardware acceleration

If the interrupt is very frequent (it would be if the serial transmit rate is high, or the serial chip buffer is small), 20 instruction cycles to save context, and another 20 cycles to restore, is a very heavy system overhead.

To alleviate this problem, shadow register sets (alternative register banks) are hardware sets of backup registers that are automatically used for interrupts (and certain other occasions).

The original registers are unchanged, so at the end of the routine, a single instruction restores the previous context. This is one instruction cycle instead of the 40 above.

The ADSP2181 has a complete set of its 19 ALU, MAC and SHIFTER registers in a single shadow bank (but the DAGs are not shadowed).

The ARM only shadows some registers, but it has 5 shadow banks (for various interrupts and exceptions) instead of the ADSPs single bank.

Hardware acceleration

Sets of shadow registers can reduce the length of time that an ISR takes to complete, so that control returns more quickly to the code that was interrupted. They also help interrupts to be serviced more quickly -many *real-time* tasks need to be serviced quickly.

One final point is why not use banks of registers to speed up multi-tasking or multi-threaded code? In fact the ARM has a special banked set of registers for a *supervisor mode* that is entered through a software command.

Hardware acceleration

The UltraSPARC II actually used register windows with each new procedure being allocated a separate window of fresh registers. When procedures are nested, the available register space is not enough, so old windows are stored to memory, and new windows take their place.

This is a type of cache for registers, and unfortunately it means there is a penalty for a register window miss (which requires a wait for the whole set of registers to be loaded from memory after the set they are replacing is first saved).

Note: The UltraSPARC also has a few global registers that are always accessible.

Hardware acceleration

The picoJAVA does not have traditional registers because JAVA programs do not require access to registers; they operate on a stack principle - most instructions (bytecodes) assume that operands are popped from the top of the hardware stack, and the result pushed back on to the stack.

However problems occur if the two wanted operands are not at the top of the stack - they have to be moved there f rst.... Performing the move wastes instruction cycles.

For this reason, the picoJAVA can also access the current top 64 stack elements as registers. If more than 64 values are popped into the stack, old entries are retired into a stack cache. This happens in the background (i.e. It does not waste any CPU cycles).

Hardware acceleration

Hardware Stack

We have considered the use of the stack, but not considered whether this was implemented in software or hardware.
In fact software stacks are often used - especially for 'C' programs. JAVA programs also require a stack (but this was implemented in hardware in the picoJAVA).

Many processors have a small hardware stack that is used to allow nested function calls. For example, the PIC16C84 has an 8-level hardware stack. Every time a function call or interrupt occurs, the PC is pushed on to the stack. To return from the function (or interrupt), a value is popped from the stack into the PC.

Hardware acceleration

Hardware stack: An 8-level deep hardware PC stack only allows 8 copies of the PC to be pushed. If the processor has a single interrupt, then 1 stack entry must be reserved for when the interrupt occurs.

There are thus 7 levels of nested function calls allowed. If too many calls occur, then the bottom address will be lost: when the pop occurs later to retrieve that address, an undefined value will be returned to the PC, and the processor will jump to an undefined memory location, causing an error.

In the PIC there are no *PUSH* and *POP* mnemonics. These are implicit in the *CALL* and *RETURN* instructions.

Hardware acceleration

Hardware stack: Although a hardware stack is a fast and eff cient method to implement function calls and returns, it limits the number of nested functions. In such a system, recursive calls can not be made.

The ADSP2181 actually has 4 hardware stacks. One is the 16-level PC stack, and the other three are *loop*, *status* and *counts*.

The *loop* stack allows 4 nested loops (i.e. do-while) to be performed in hardware, whilst the *count* stack allows another 4 nested count-based loops (i.e. For-next) to be accommodated in hardware.

The *status* stack allows interrupts to be nested in a similar way to the shadow status registers of the ARM. In the ARM there are dedicated shadow status registers, but the ADSP has an *n*-level status stack, where *n* is the number of possible interrupt sources.

Branch Prediction

Pipelined CPUs in general suffer from three major issues:

1. Interrupt response can be poor

RISC processors signif cantly improve on interrupt response over their CISC predecessors, but with longer pipelines, interrupt response worsens.

2. Code dependencies limit performance

We will look at this more – consider the implications for both machine architecture and compiler design!

3. They are only good for continuous code

Short programs, and lots of branching reduce pipeline eff ciency.

Branch Prediction

Branching causes problems*:*

```
i1:          ADD   R1, R2, R3
i2:          B     loop1
i3:          ADD   R0, R2, R3
i4:          AND   R4, R2, R3
i5:  loop1   STR   R1, locationA
```

Clock cycle:	1	2	3	4	5	6
Fetch	i1	i2	i3	i5		
Decode		i1	i2	**X**	I5	
Execute			i1	i2	**X**	i5
Save				i1	i2	**X**

By the time we decode i2 and discover it's a branch, we've already incorrectly fetched i3.

So we have to immediately **stall the pipeline** until we fetch i5 instead.

Branch Prediction

Branching can cause even worse problems:

```
i1:          ADD   R1, R2, R3
i2:          B     loop1
i3:          ADD   R0, R2, R3
i4:          AND   R4, R2, R3
i5:  loop1   STR   R1, locationA
```

In most RISC processors the branch target address is stored as a **relative offset from the current address**.
Thus in assembler, *i2* above would be encoded as “B +3”

Before the branch can be taken, the **absolute** target address needs to be calculated first:

PC <- PC + offset

Branch Prediction

In many processors, the branch target address calculation is performed using the main ALU. Higher performance processors might have a dedicated address ALU for this. Let's look at the normal situation:

```
i1:          ADD   R1, R2, R3
i2:          B     loop1
i3:          ADD   R0, R2, R3
i4:          AND   R4, R2, R3
i5:  loop1   STR   R1, locationA
```

3 stage scalar pipeline F – E – S

Clock cycle:	1	2	3	4	5	6
F & D	i1	i2	**X**	i5		
E (ALU)		i1	i2	**X**	-	
E (MUL)		-	-	**X**	-	
E (MEM)		-	-	**X**	i5	
S			i1	-	**X**	i5

Note: even though i2 uses the ALU, the result does not need to be saved (it is not going to a register). Instead, it is *forwarded* directly to the fetch unit.

Branching causes **X**, a **pipeline stall**

Branch Prediction

Conditional Branching causes even more dependencies

```
i1:          ADDS R1, R2, R3
i2:          BEZ     loop1
i3:          ADD  R0, R2, R3
i4:          AND  R4, R2, R3
i5:  loop1   STR  R1, locationA
```

4 stage scalar pipeline F – D – E – S

The branch needs to know if the condition is fulfilled before it can be taken (as well as waiting for a branch address calculation). Here it will sit and wait (**S**tall) until it knows:

Clk cycle:	1	2	3	4	5	6	7	8
F	i1	i2	i3	**S**	i5			
D		i1	i2	i3	**S**	i5		
E (ALU)			i1	i2	**S**	**S**	-	
E (MUL)			-	-	**S**	**S**	-	
E (MEM)			-	-	**S**	**S**	i5	
S				i1	-	**S**	**S**	i5

BRANCH TAKEN

Clk cycle:	1	2	3	4	5	6	7	8
F	i1	i2	i3	**S**	i4	i5		
D		i1	i2	i3	**S**	i4	i5	
E (ALU)			i1	i2	i3	**S**	i4	-
E (MUL)			-	-	-	**S**	-	-
E (MEM)			-	-	-	**S**	-	i5
S				i1	-	i3	**S**	i4

BRANCH NOT TAKEN

Branch Prediction

What can be done to reduce the branch penalty

1. Make **all** instructions conditional (ARM)

2. Separate condition setting and conditional

3. Pre-calculate branch target address

4. Use a delayed branch (MIPS, TMS320C5x etc...)

5. Allow speculative execution

Refer to the text for more explanation of these!

Parallel Machines

We will look at parallel machines later, but actually there are several levels of parallelism that can be considered in computers, since the term “parallel machines” is very loosely de ned:

Bit-level parallelism relates to the size of word that a computer processes. An 8-bit computer processes eight bits in parallel, but four times as much data can potentially be handled in a 32-bit CPU.

Instruction level parallelism. As we have seen in many cases, different instructions can be overlapped and processed simultaneously, providing we take care of any dependencies between them. Pipelining is a simple example, but superscalar machines, also co-processors and Tomasulo’s Algorithm.

Vector parallelism relates to SIMD machines that process not just single words of data, but entire vectors at one time. SSE and MMX are examples of this type of parallelism.

Task parallelism means that entire tasks, or program subroutines and functions, can be executed simultaneously by different hardware.

Tomasulo's Algorithm

One of the best examples of scheduling in hardware is **Tomasulo's Algorithm**

Although this is ~40 years old (IBM 360/Model 91), it's much more interesting than most of the modern examples!

Apart from that, it can yield the highest possible performance of a hardware scheduler, since it allows out-of-order execution and out-of-order completion.

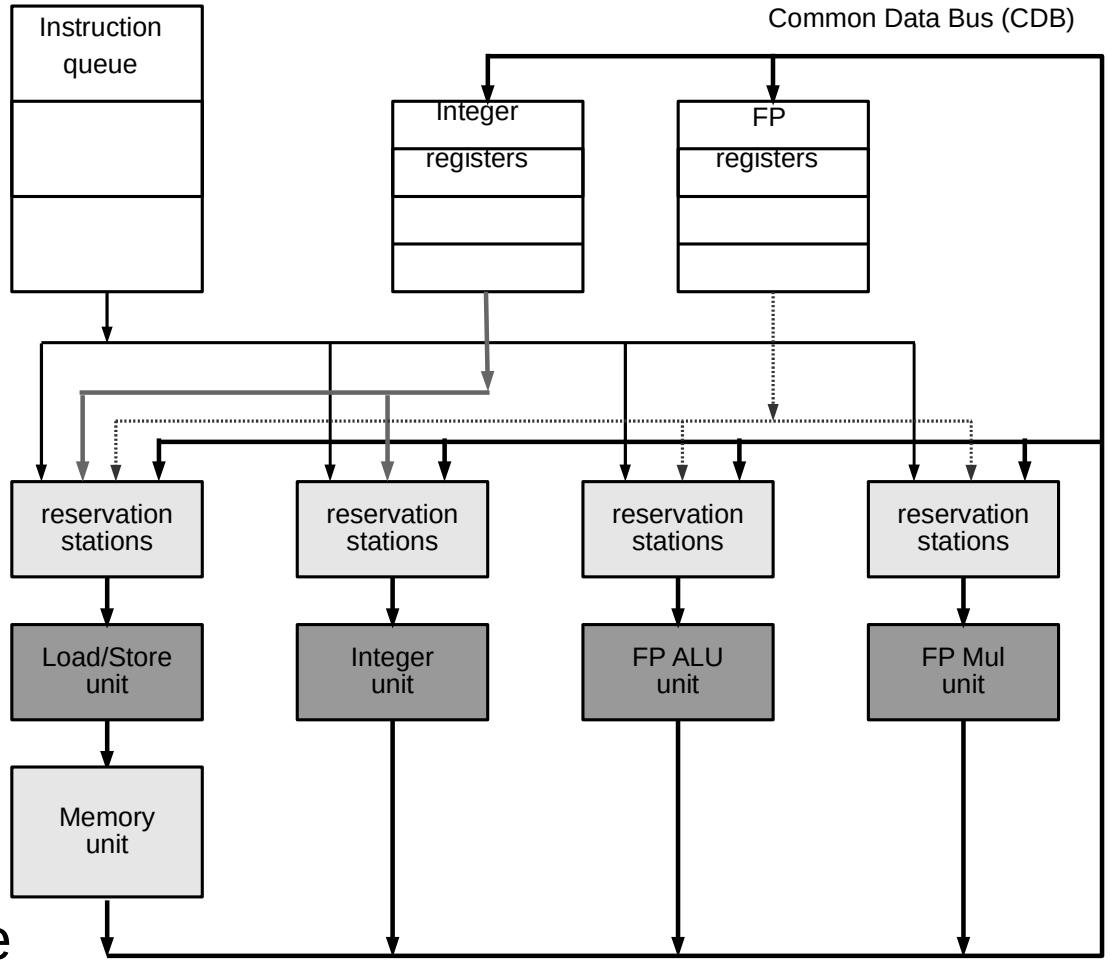
Tomasulo's Algorithm

The pipelined functional units can be used independently, but take different lengths of time to complete.

The instruction queue is fed from main memory.

The common data bus conveys completed results

The reservaton stations hold instructions waiting for space in the functional units and maybe waiting for their operands too.

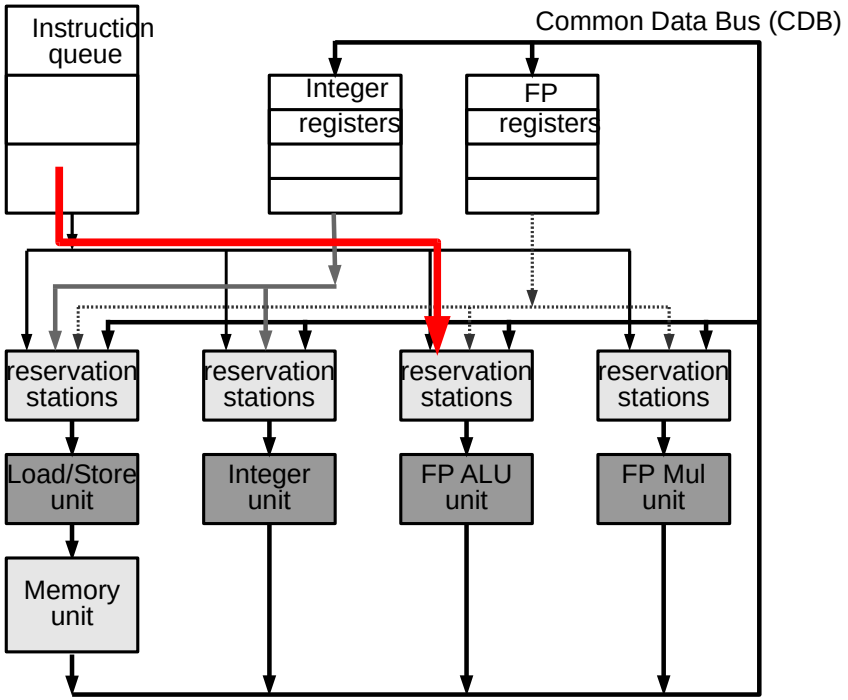


Tomasulo's Algorithm

ISSUE

Instructions in the queue are issued in-order to the relevant RS when they become free.

Inside the RS, the operands are co-located with the instruction. These are either real (i.e. they came from a register) or virtual (i.e. are waiting for a previous result), and this is decided when the instruction is issued.



If the required RS is not free, the instruction queue stalls.

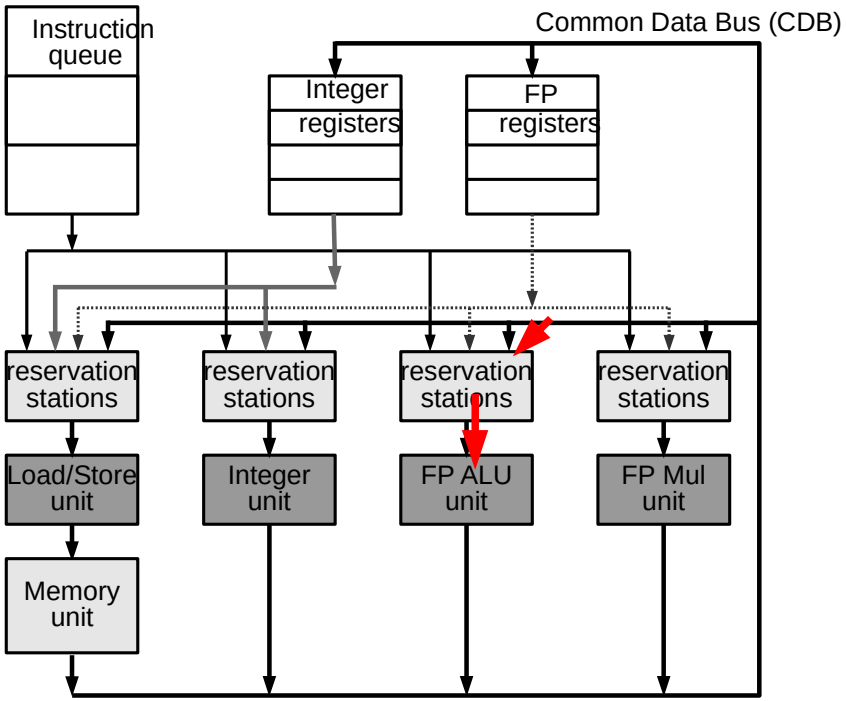
Tomasulo's Algorithm

EXECUTE

The RS will feed instructions to the functional units with all the operands are available and the functional unit is free.

Instructions with real operands are immediately ready to be executed.

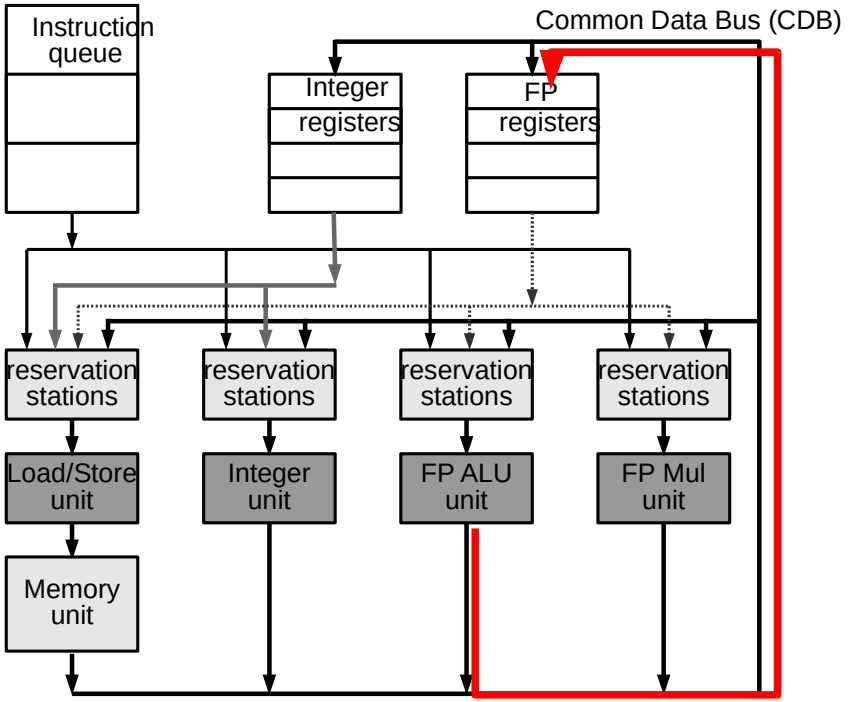
For instructions with virtual operands, the RS must look on the CDB until it finds the correct value (the result of a previous calculation).



Tomasulo's Algorithm

COMPLETION

Results are written back to the register bank, and to any waiting RSs using the CDB (memory writes are written to memory).



Tomasulo's Algorithm

In Tomasulo, the instructions are issued sequentially to the RSs, so mode changes, interrupts and so on happen as in a statically pipelined machine.

The use of virtual values (i.e. operands waiting for a result) allows instructions to be issued to RSs which would block a normal pipeline.

Multiple-slot RSs are used to collect all information required by an instruction, then execute as soon as all the information is there: opcode + operands.

This resolves both structural and data dependencies...

Conclusion

There are lots of ways to make a faster CPU apart from just increasing the clock speed. We can do more per clock cycle with more powerful instructions, allowing >1 instruction to execute in parallel. We can also overlap instructions in a pipeline – and if so can take some special actions to make the pipeline faster and more eff cient (including branch prediction, delayed branches etc...)

We can also create specialised co-processors for data handling, but also handle instructions faster, and code structures like loops, addressing modes, conditional instructions and register handling for both normal code and for interrupt service routines.

New CPU designers have many options, and a lot of f exibility!