Lecture 5

Interrupt, Pipeline, and Encoding

Interrupts

- Many applications for microprocessor systems require things to happen when an event occurs unexpectedly.
- These unexpected events can interrupt the normal operation of the microprocessor.
- During an ‘interrupt’ the execution of the main program is halted and a special program is executed instead.
- When the interrupt program has finished, the microprocessor returns to the main program.

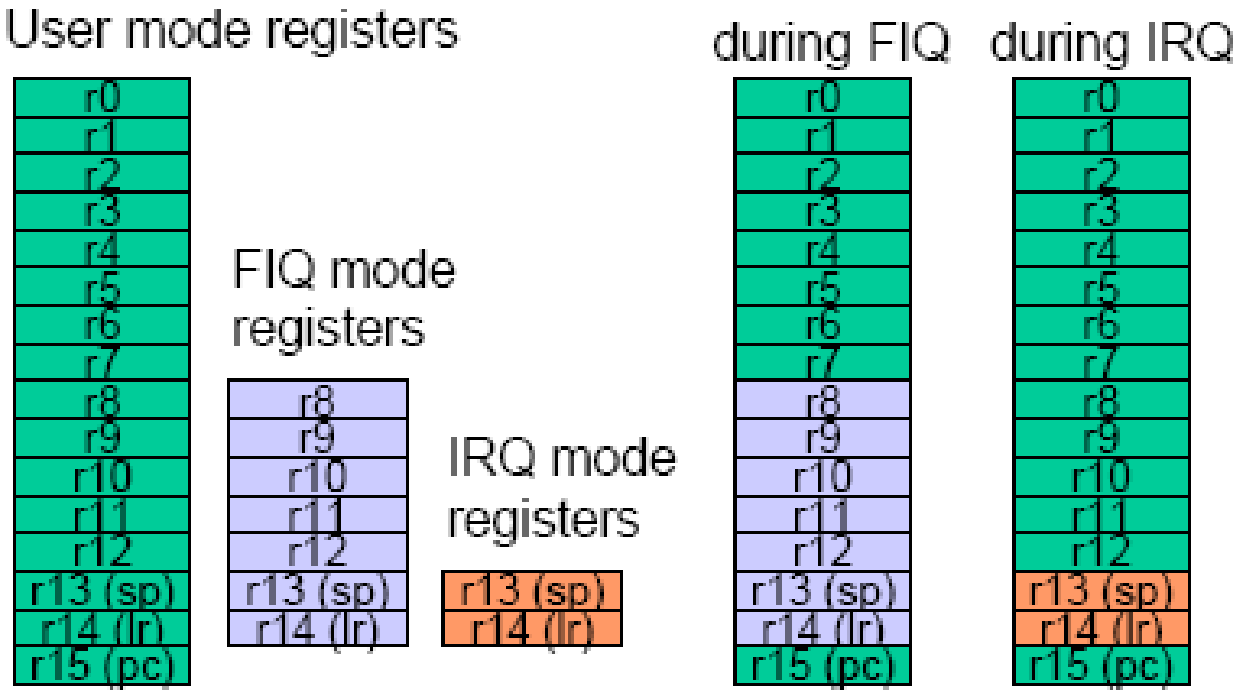
Interrupts

- Interrupts provide a very convenient method for dealing with events and this method is often used for events that are not unexpected
  - e.g.
    - when a mobile phone receives an incoming call.
- An interrupt is triggered when the microprocessor receives a (voltage) signal on a special connection within the control bus.
- The ARM7 has two types of interrupts,
  - a normal interrupt (IRQ) and
  - a fast interrupt (FIQ).

Interrupt Handling

- What happens when the microprocessor receives an interrupt signal?
- The microprocessor switches ‘mode’.
- The ARM normally operates in ‘user mode’
  - when a normal interrupt is received it switches to ‘IRQ mode’ and
  - when a fast interrupt is received it switches to ‘FIQ mode’.
- Each mode has it’s own link register and stack pointer.
- In addition FIQ mode has it’s own registers r8 to r12.

Registers in IRQ and FIQ modes

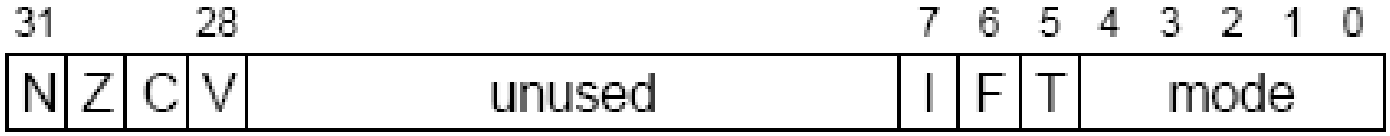


Current Program Status Register

- CPSR is used in user-level programs to store the condition code bits
- The user-level programmer needs not usually be concerned with how this register is configured.

ARM CPSR format

Current program status register



User:	10000
FIQ:	10001
IRQ:	10010
SVC:	10011
Abort:	10111
Undef:	11011
System:	11111

Interrupt Handling

- When an interrupt occurs the following happens:
  ① The registers for IRQ mode or FIQ mode are activated.
  ② The current program status register, CPSR, (this contains the flags and other information) is saved into a saved program status register, SPSR.
     a) There are two SPSR registers, one for each mode.
  ③ The return address of the next instruction to be executed in the main program is stored in the link register for the appropriate mode.
  ④ The program counter is set to either 0x00000018 for an IRQ or 0x0000001C for a FIQ.

Interrupt Vectors

- The first instruction executed by an interrupt program is the instruction stored at memory address 0x00000018 for an IRQ or 0x0000001C for a FIQ.
- These memory addresses are known as ‘vectors’.
- For an IRQ the instruction at address 0x00000018 must be a branch to another part of memory because the following memory location, 0x0000001C contains the first instruction of the FIQ handler.

Returning from Interrupts

- When the program called by the interrupt has finished, the microprocessor returns to the main program. This is achieved by doing the following:
  - The saved program status register is copied back into the current program status register.
  - The link register in the IRQ mode or FIQ mode is copied into program counter.
  - The microprocessor returns to the mode it was in before the interrupt occurred - normally user mode.

FIQ or IRQ ?

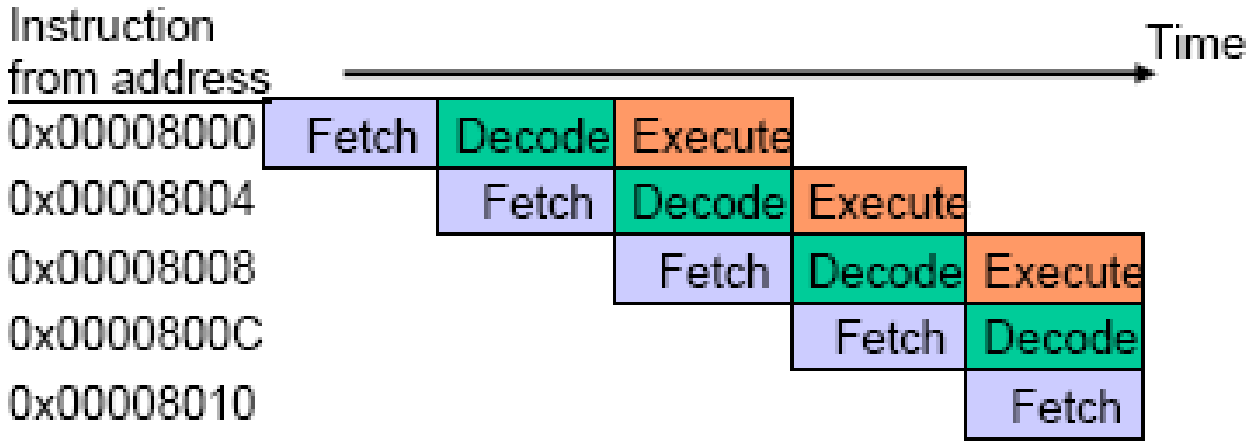
- Generally the most important interrupt is assigned to the FIQ and all other interrupts are assigned to IRQ.
- There are two reasons for this:
  - IRQ is disabled by an FIQ and if a FIQ and an IRQ occur simultaneously the FIQ is serviced first.
  - FIQ can be serviced as quickly as possible because
    a) there is no need to branch as for an IRQ and
    b) normally user mode registers are pushed onto the stack when an interrupt occurs so that they are not corrupted but for an FIQ there is no need to stack registers r8 to r12.

Instruction Pipelines

- Instruction pipelines are an important feature of all modern microprocessors.
- For the same basic speed of transistor operation, an n stage instruction pipeline allows the microprocessor to execute up to n times as many instructions in a given time.
- The ARM7 microprocessor has a three stage pipeline
  - one stage for each of the CPU cycles;
    - fetch, decode and execute.

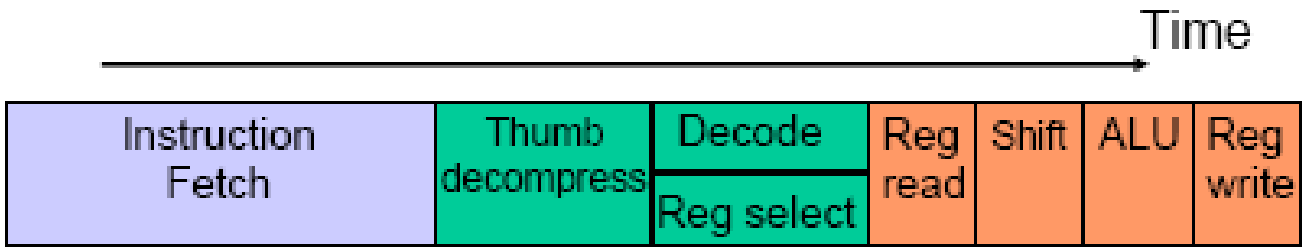
Instruction Pipelines

- In a three stage pipeline, the CPU can simultaneously execute an instruction, decode the next instruction and fetch the next instruction.



ARM7 3 stage pipeline: detail

In each stage of the ARM7 pipeline, several things happen; normally consecutively:



FETCH
Instruction fetched from memory.

DECODE
Thumb to ARM decompression.
Decode.
Register selection

EXECUTE
Register read.
Barrel shift.
ALU operation.
Register write.

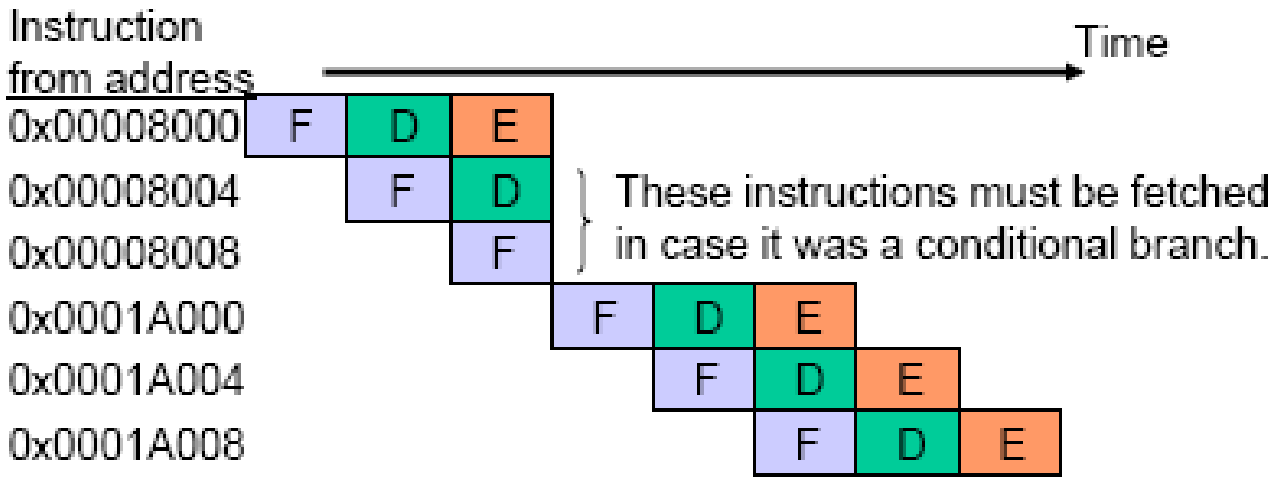
For Internal Use Only!

Pipelines - optimum operation

- A pipeline operates optimally if the instructions to be executed are in consecutive memory locations and no conflict occurs on the data bus.
- When this is the case, the microprocessors operates at one clock cycle per instruction (1 CPI).
- The best performance cannot be achieved if a branch or load instruction is executed or if an interrupt is serviced.

Branch Instruction

- A branch instruction will reload the program counter so that two cycles are lost
  - e.g. assume that the instruction at address 0x00008000 is Branch to 0x0001A000

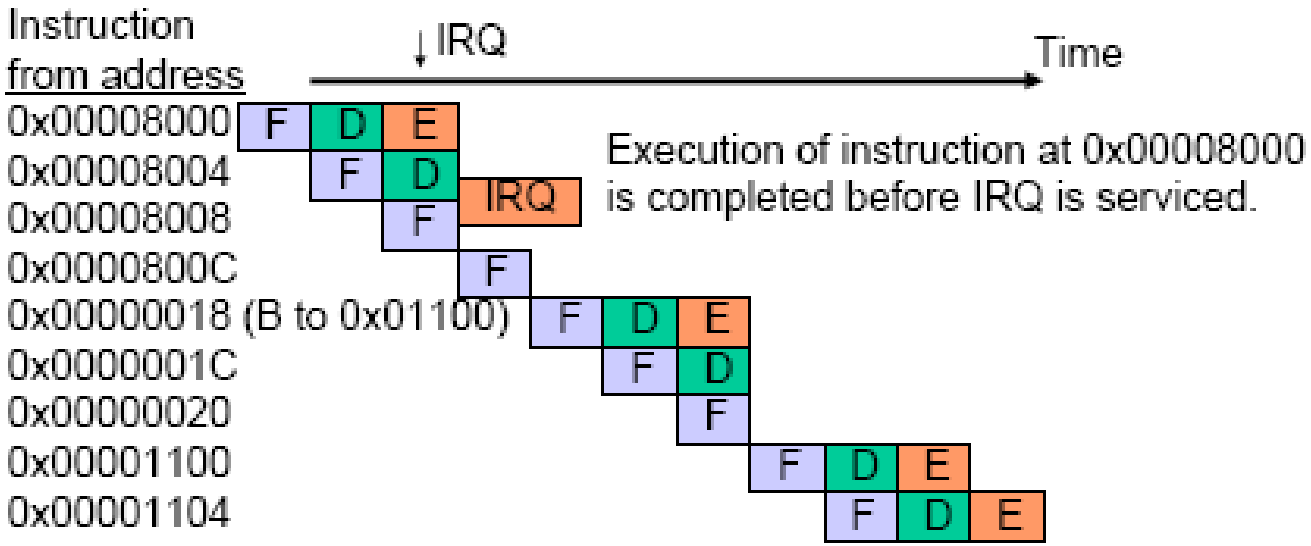


Branch Instruction

- So in six clock cycles only four instructions are executed – 1.5 CPI (*count from the execute stage of the first instruction to the execute stage of the last instruction*).
- For a branch and link instruction the link register is updated during the two clock cycles when no instruction is being executed.
- If a conditional branch is not executed then no clock cycles are lost.

Interrupts

An IRQ reloads the program counter with 0x00000018 and then branches so that the pipeline is broken twice.

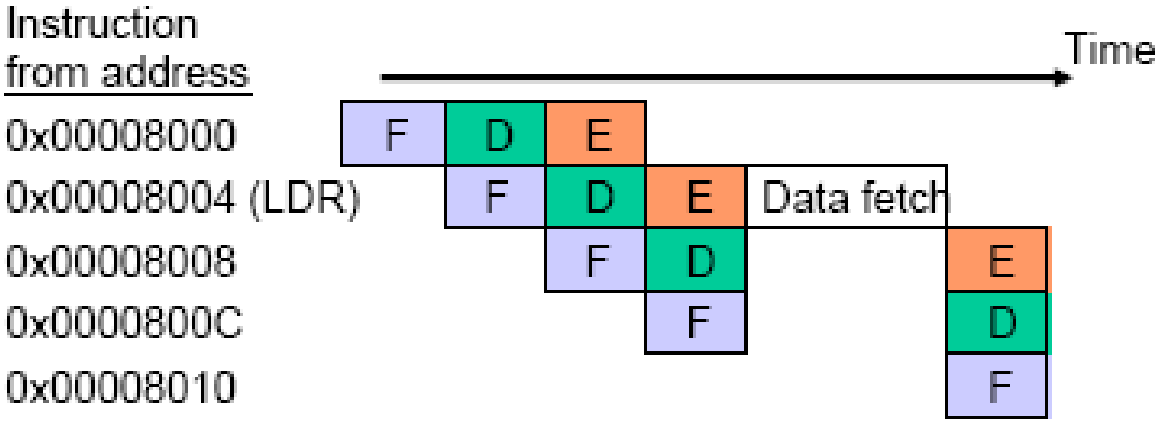


Interrupt latency

- The latency is the time between the microprocessor receiving an interrupt signal and the first instruction being executed (i.e. the instruction at 0x00001100 in example).
- The minimum latency for an IRQ is 7 clock cycles
  - minimum because an IRQ could be interrupted by an FIQ.
- The minimum latency for an FIQ is 4 clock cycles because instructions can start from address 0x0000001C and there is no need to branch.
- FIQ can be interrupted by a system reset.

Load and store instructions

- Load and store instructions use the data bus to pass data to or from memory so that the data bus cannot be used to fetch an instruction at the same time.



Eliminating bus conflicts

- The data fetch uses two clock cycles so that in the previous example only three instructions are executed in five clock cycles - 1.66 CPI.
- Bus conflicts could be eliminated by using two separate data buses; one for instructions and one for load and store data - this is called a Harvard architecture.
- A microprocessor, such as the ARM7, that uses one data bus for both instructions and data is said to have the von Neumann architecture.

Von Neumann or Harvard?

- The advantage of Harvard over von Neumann is that there are fewer lost clock cycles due to bus conflicts.
- The drawback is greater complexity.
- The ARM9 (next generation after ARM7) has a Harvard architecture and a five stage instruction pipeline.
- The ARM7 has an average CPI of 1.9 whereas the ARM9 has an average CPI of 1.5.
  - Note that this depends upon the program being executed but typical data processing programs make a lot of use of load and store.

3 stage or 5 stage pipeline?

- The maximum clock frequency of the ARM7 is limited by a bottleneck during the execute stage of the 3 stage pipeline.
- The ARM9 has a 5 stage pipeline so that there is less work in each stage of the pipeline and therefore a higher maximum clock frequency can be achieved for the same technology.
- However more stages require more power because more circuits are functioning simultaneously.

ARM7 pipeline example

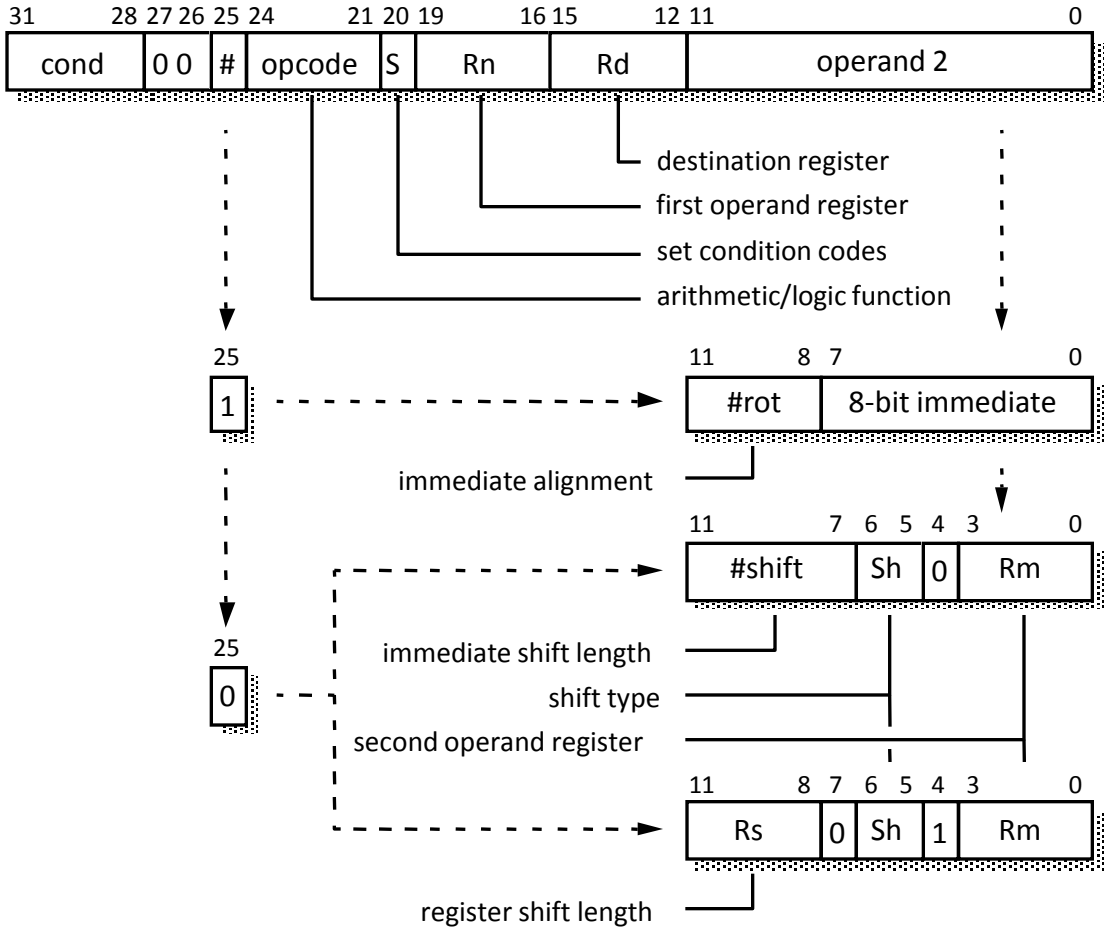
Consider a typical three operand ARM7 instruction e.g.

ADDNE r11, r3, r7 LSR #8

Add the contents of register r3 to the contents of register r7 shifted right by 8 bits and put the sum in register r11 if the zero flag is clear (Z=0).

The machine code for this instruction is 0x1083B427

Data processing instruction binary encoding



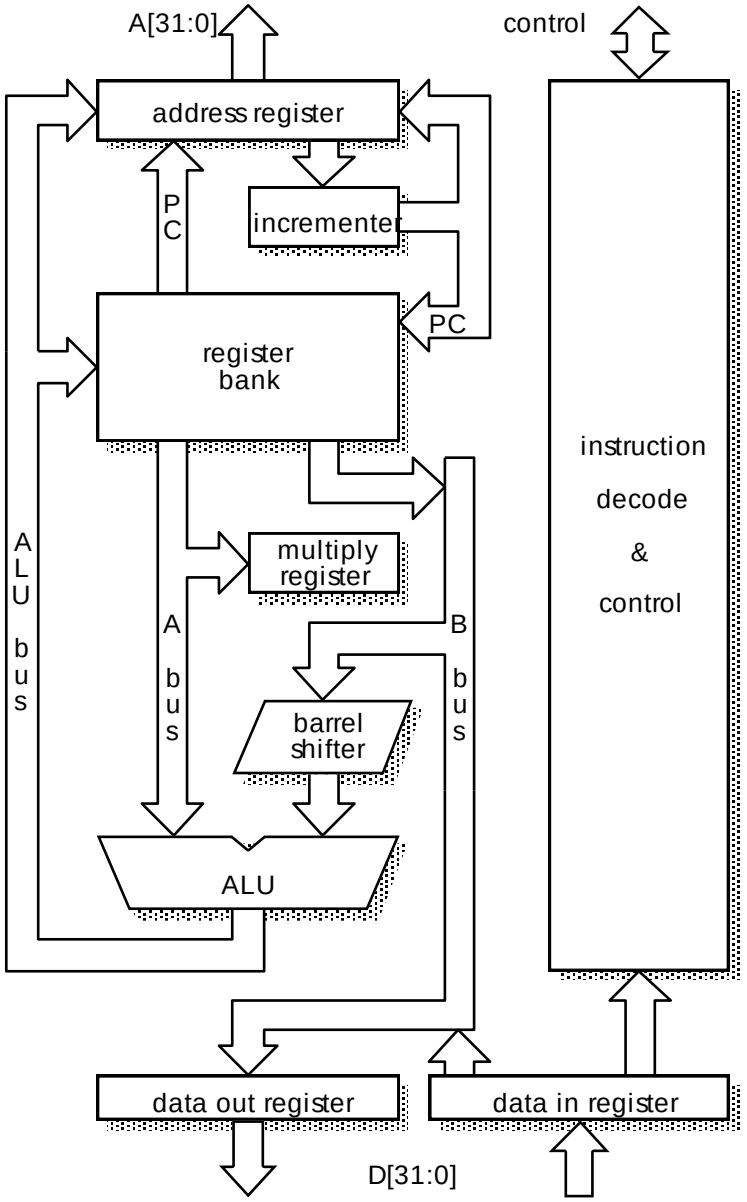
For Internal Use Only!

ARM data processing instructions

Opcode [24:21]	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd := Rn AND Op2
0001	EOR	Logical bit-wise exclusive OR	Rd := Rn EOR Op2
0010	SUB	Subtract	Rd := Rn - Op2
0011	RSB	Reverse subtract	Rd := Op2 - Rn
0100	ADD	Add	Rd := Rn + Op2
0101	ADC	Add with carry	Rd := Rn + Op2 + C
0110	SBC	Subtract with carry	Rd := Rn - Op2 + C - 1
0111	RSC	Reverse subtract with carry	Rd := Op2 - Rn + C - 1
1000	TST	Test	Scc on Rn AND Op2
1001	TEQ	Test equivalence	Scc on Rn EOR Op2
1010	CMP	Compare	Scc on Rn - Op2
1011	CMN	Compare negated	Scc on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd := Rn AND NOT Op2
1111	MVN	Move negated	Rd := NOT Op2

For Internal Use Only!

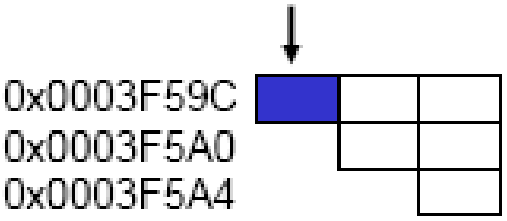
ARM organization

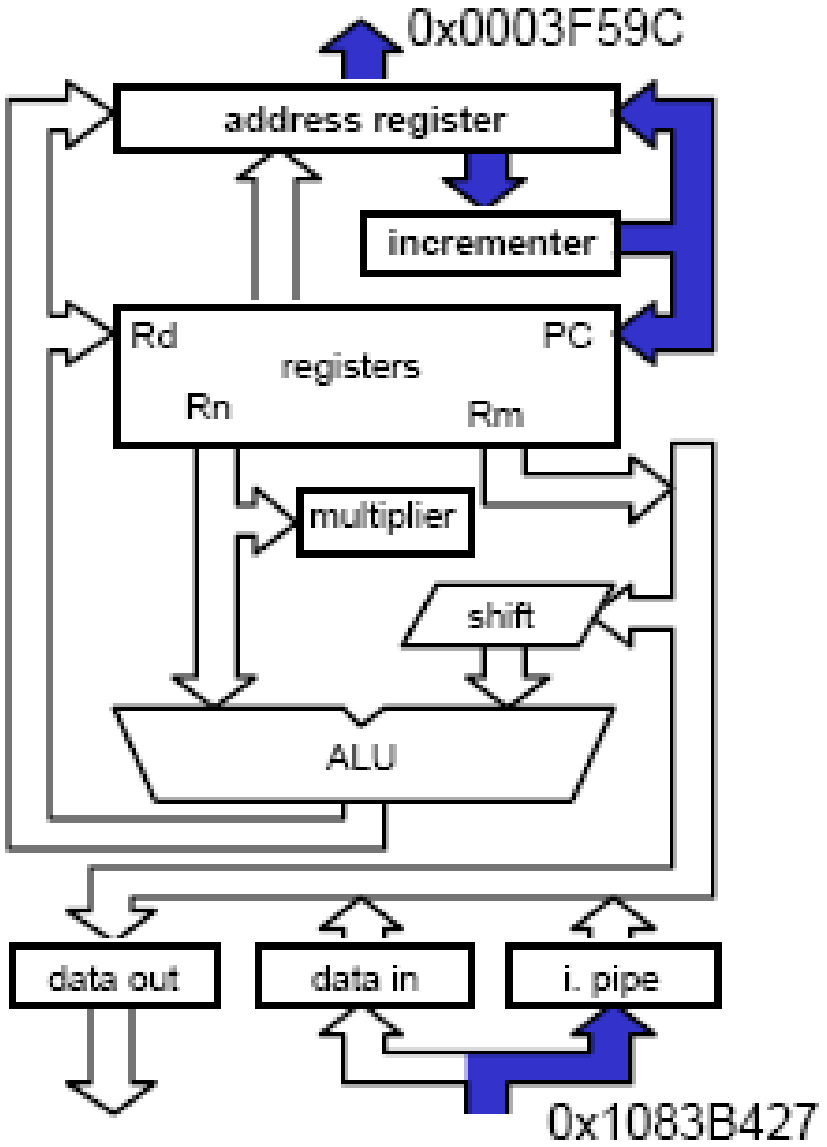


FETCH

The instruction, 0x1083B427, is fetched from a memory location with address given by the value in the address register, e.g. 0x0003F59C.

Then 4 is added to address register and it is copied to the program counter, PC = r15.

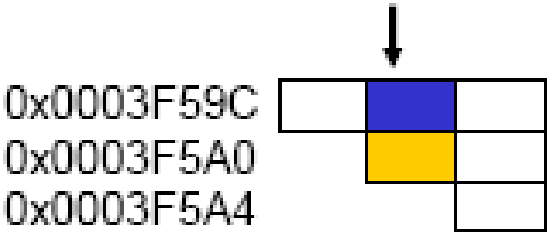


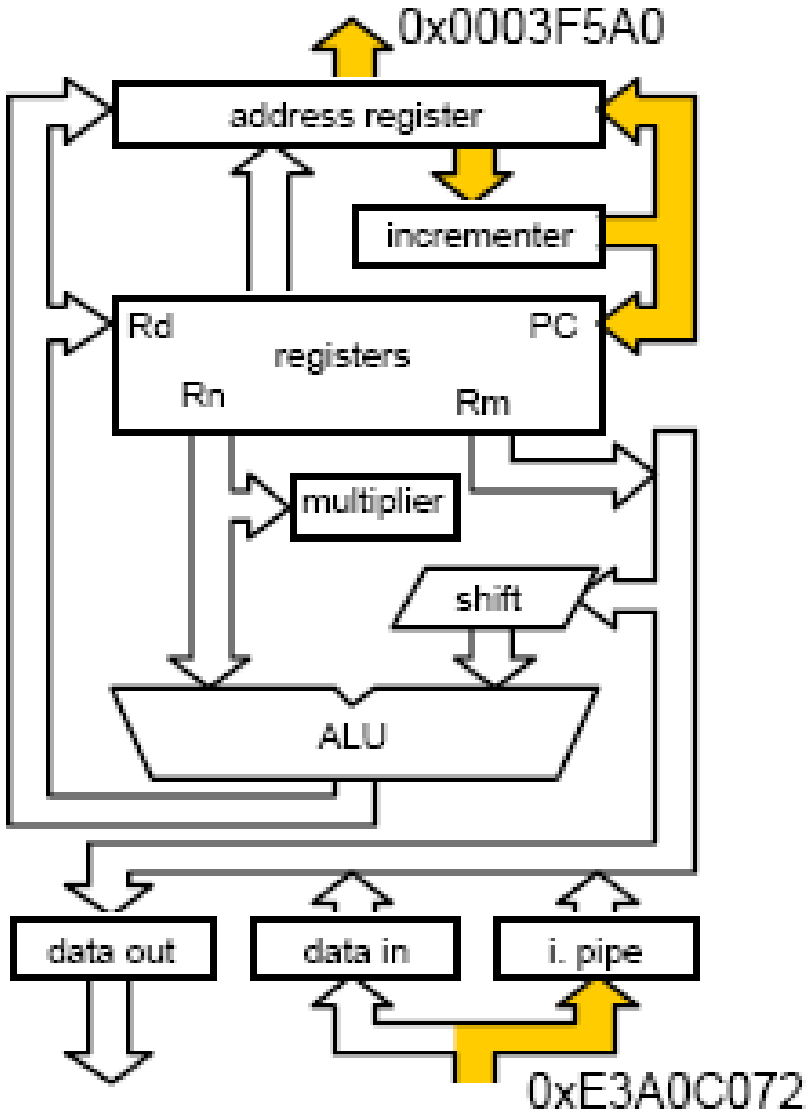


DECODE

The first action of the decode cycle only applies to 16 bit Thumb instructions.

Thumb code is converted (decompressed) into 32 bit ARM code if the processor is in ‘Thumb mode’.



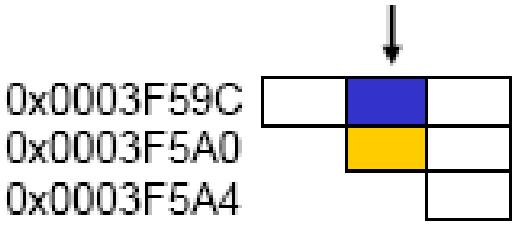


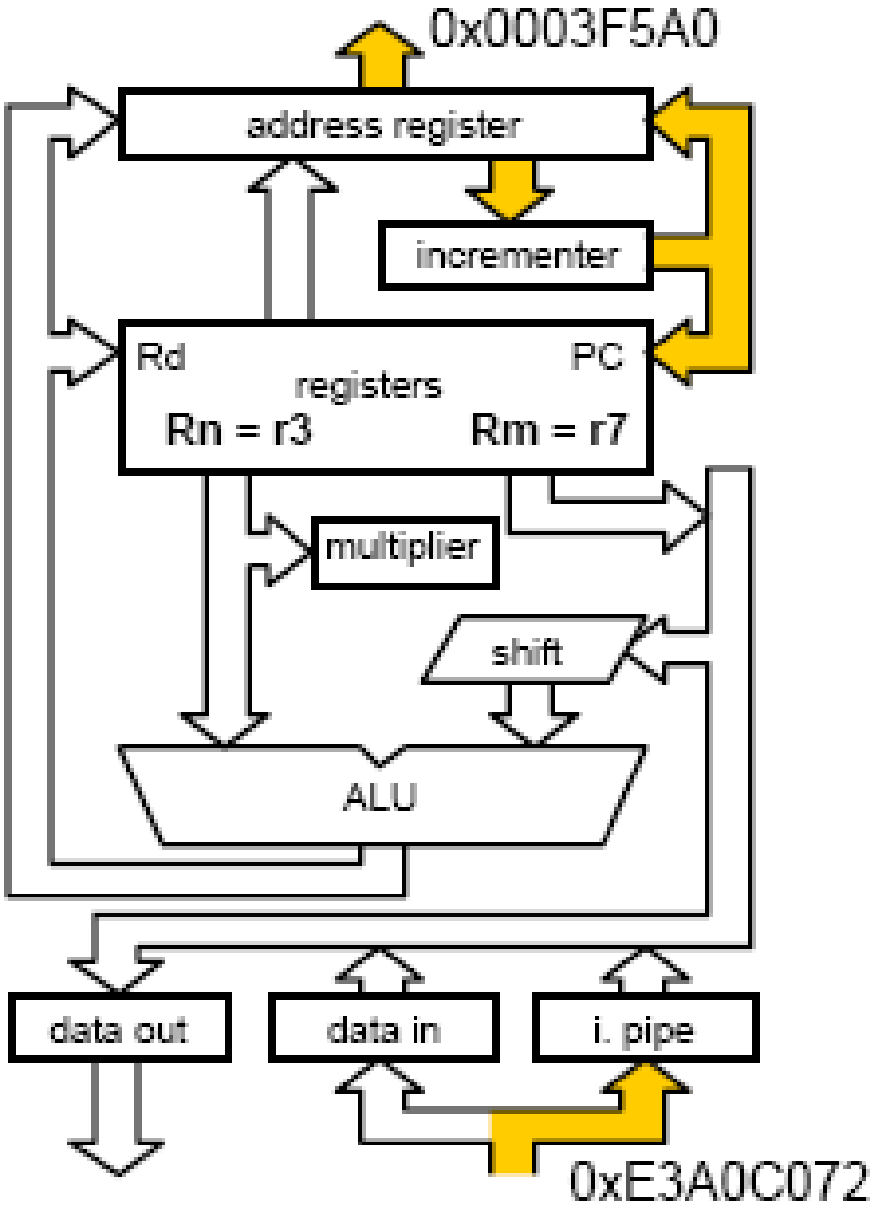
DECODE

In 2nd part of the decode cycle, the instruction, 0x1083B427, is decoded.

The condition field, NE, is compared with flags to see if the instruction is to be executed.

And the two source registers, R_n = r3 and R_m = r7, are identified.

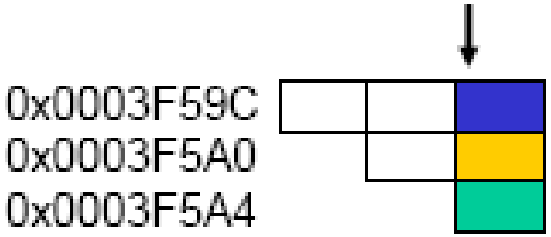


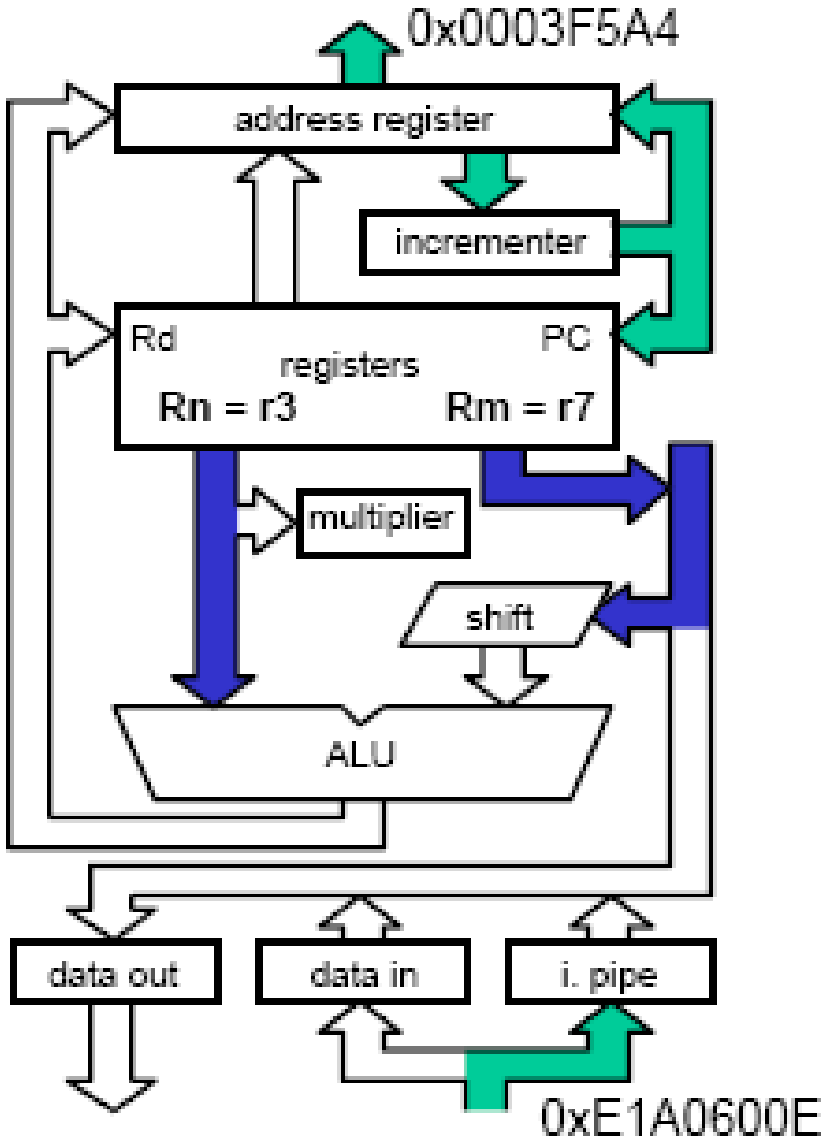


EXECUTE

Register read.

The values held in the two source registers are read onto the internal buses e.g. content of r3, 0x0F60E457, onto the A bus & content of r7, 0x3D0713A9, onto the B bus.



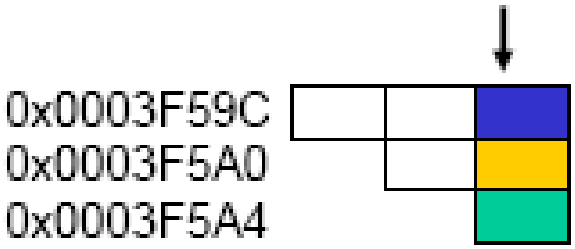


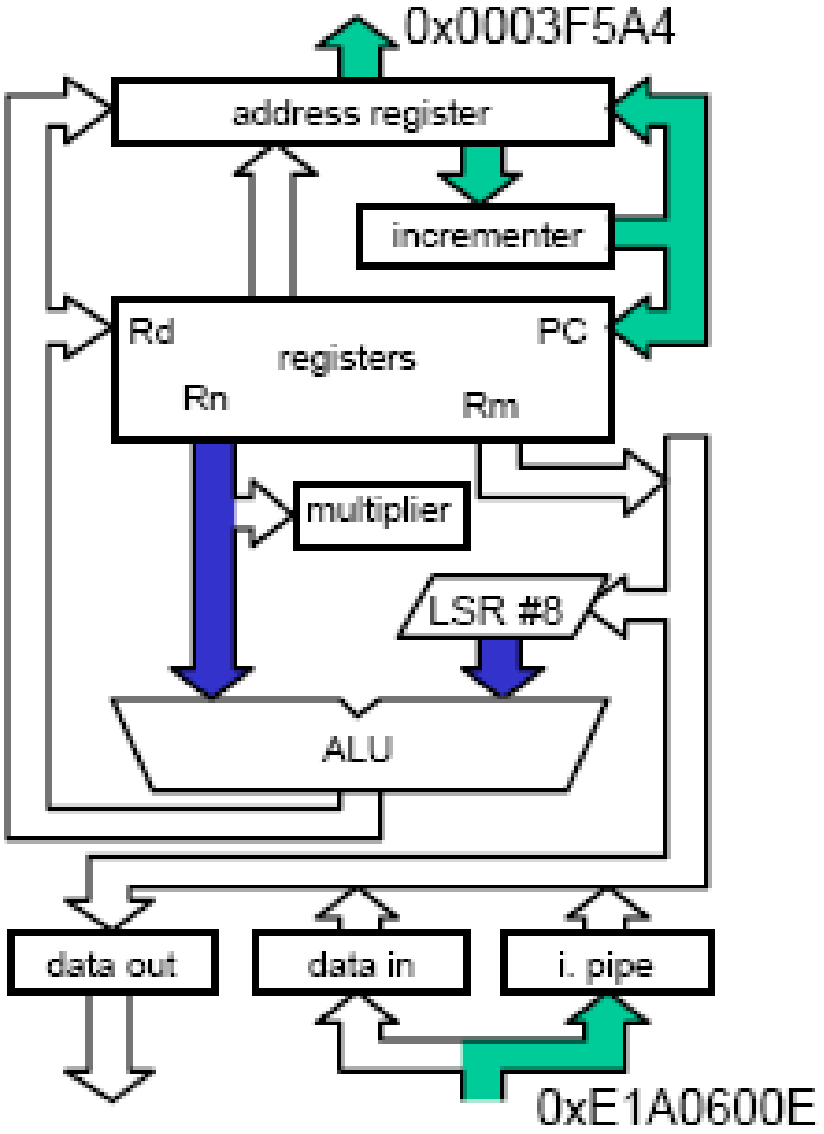
EXECUTE

Barrel shift.

The appropriate shift, e.g. LSR #8, is applied to B bus value, 0x3D0713A9.

Output of the barrel shifter, 0x003D0713, is connected to the second input of the ALU.



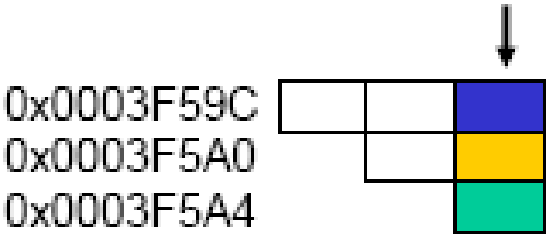


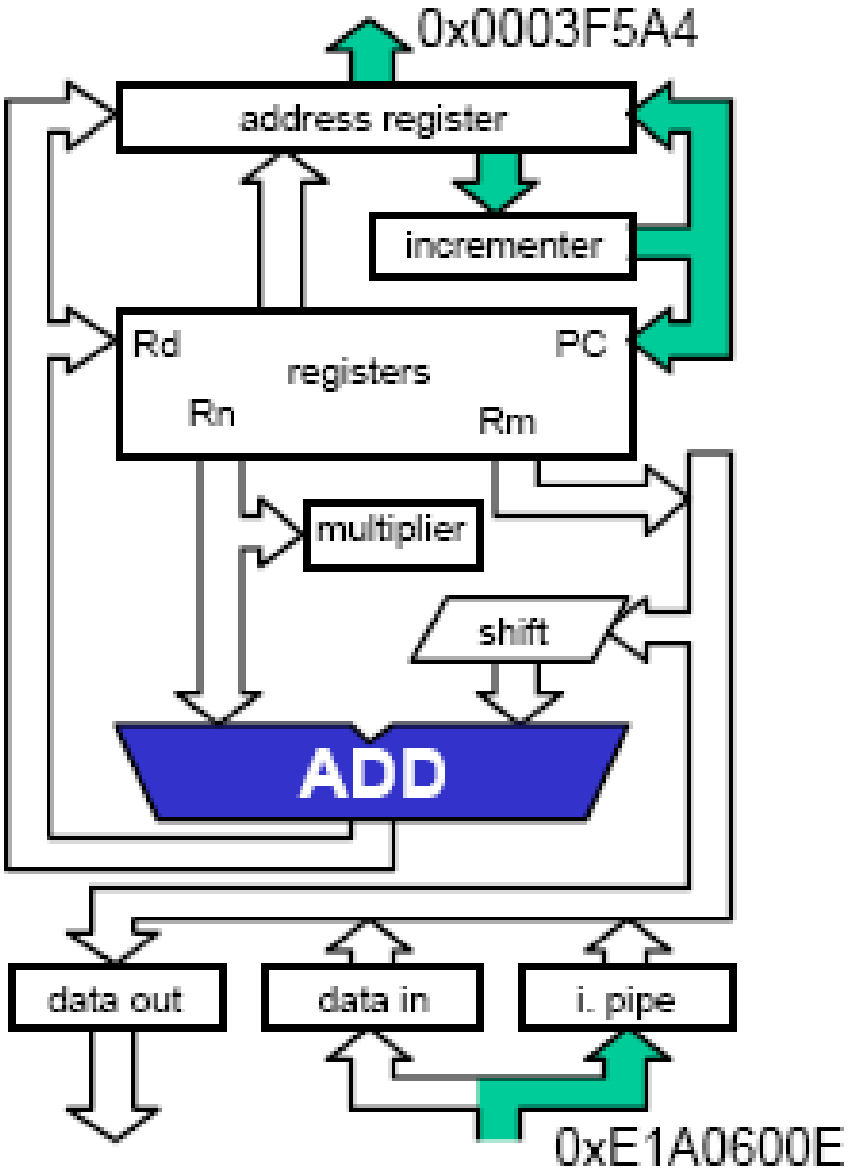
EXECUTE

ALU

The function required by the instruction, e.g. ADD, is performed by the ALU.

The output of the ALU, 0x0F9DEB6A, is connected to the ALU bus.

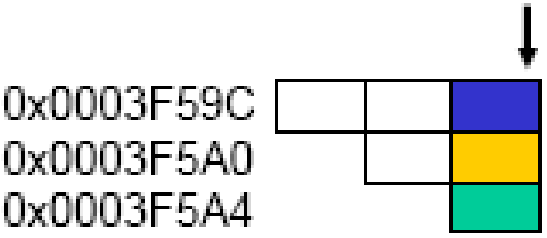


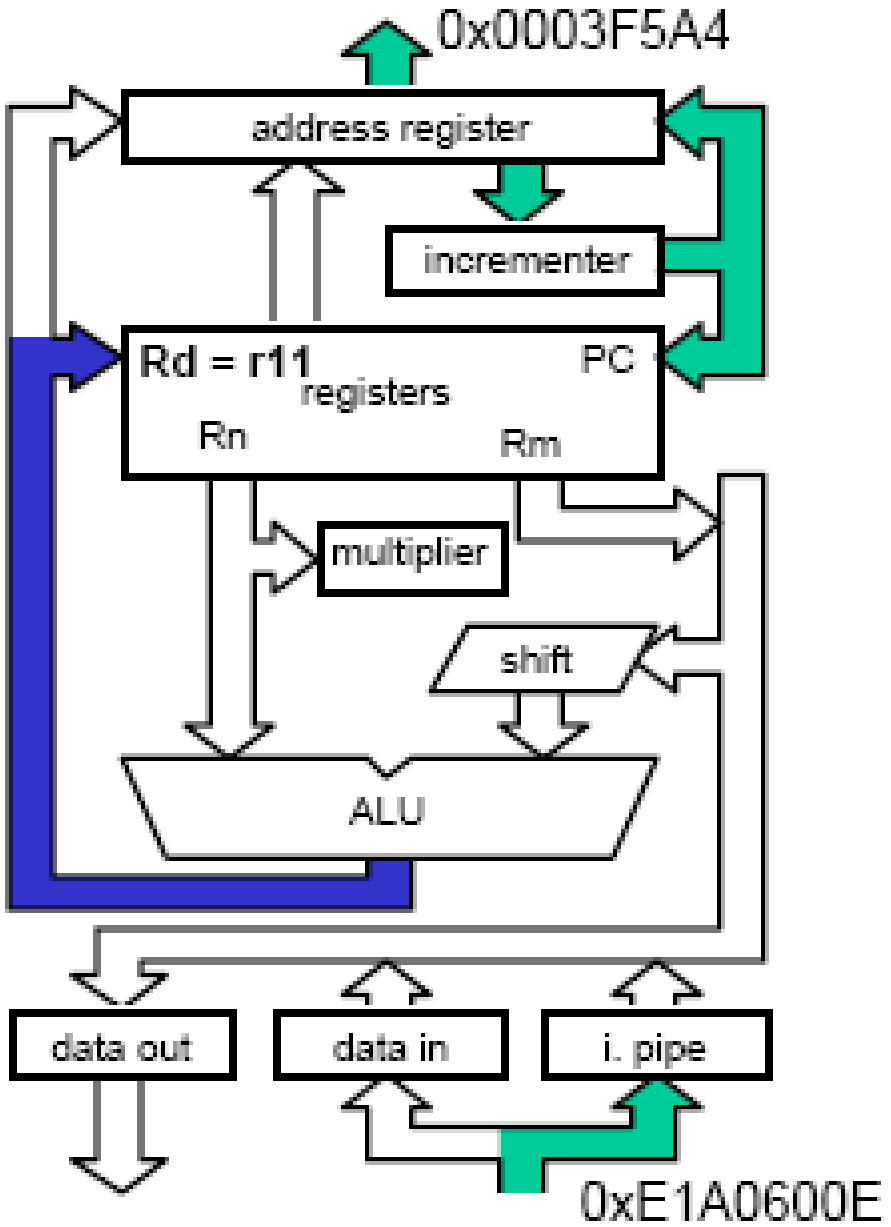


EXECUTE

Register write.

The value on the ALU bus, 0x0F9DEB6A, is written into the destination register, r11.





Program counter: r15

Note that the address register and program counter (that is register r15 in the register bank) hold a value, 0x0003F5A4, which is 8 greater than the memory address, 0x0003F59C, of the instruction that is being executed at that time.

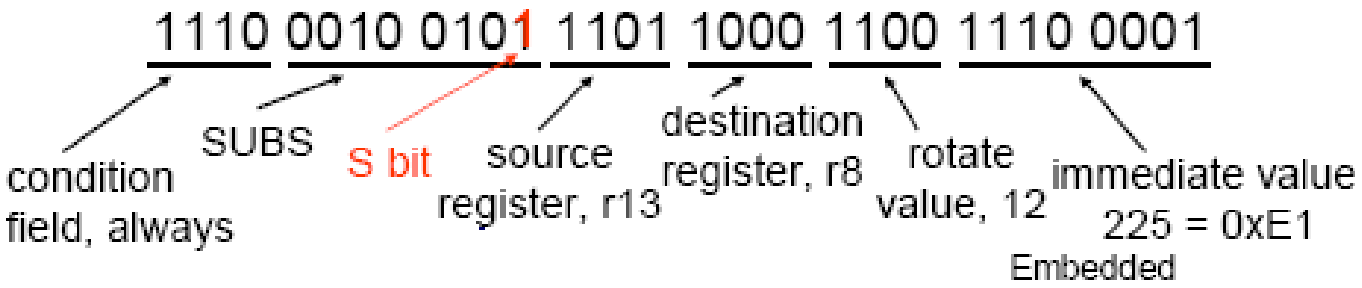
Also note that the ALU is not used to increment the program counter and address register so that it is free to execute instructions.

Another example: immediate

Consider an ARM7 instruction that uses an immediate e.g.

SUBS r8, r13, #57600

- Subtract 57,600 from the contents of register r13 and put the difference in register r8.
- Set the flags appropriately.
- Note that 57,600 = 0xE100 = 225 × 256 = 225 × $2^{2x(16-12)}$
- The machine code for this instruction is 0xE25D8CE1



Allowed immediate values

The immediate value can only be one byte (8 bits) but it does not have to be the least significant byte e.g. 57,600 = 0xE100 is allowed.

The immediate value can be any value given by $N \times 2^{2x(16-M)}$ where N is in the range 0 to 255 and M is in the range 0 to 15.

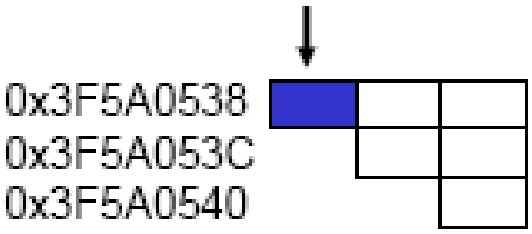
The instruction code for SUBS rZ, rY, #0xNN × 22 × (16-M) is 0xE25YZMNN.

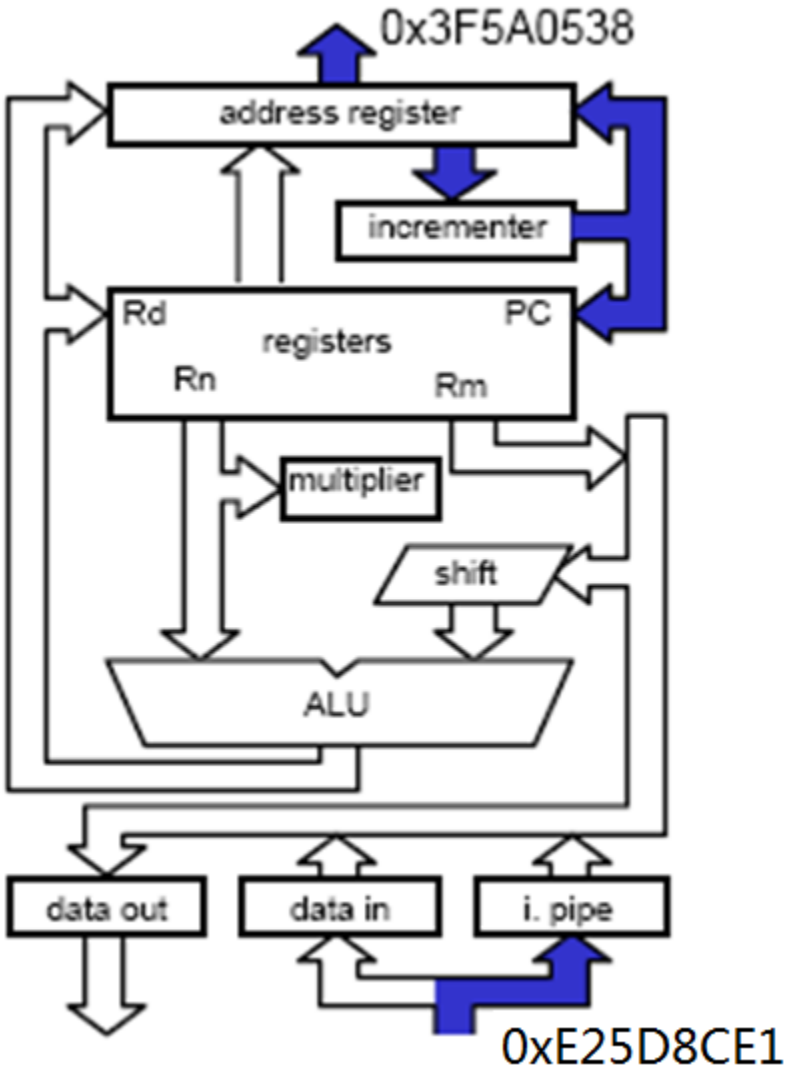
FETCH

The instruction, 0xE25D8CE1, is fetched from a memory location with address given by the value in the address register, e.g. 0x3F5A0538.

4 is added to address register and copied to PC = r15.

Decode cycle is the same as before.

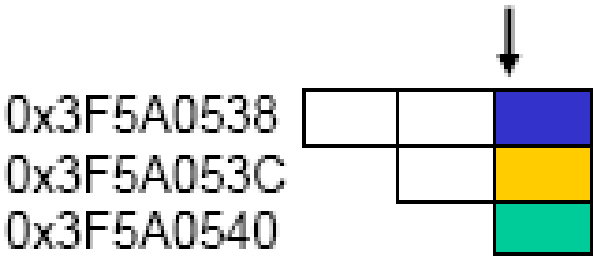


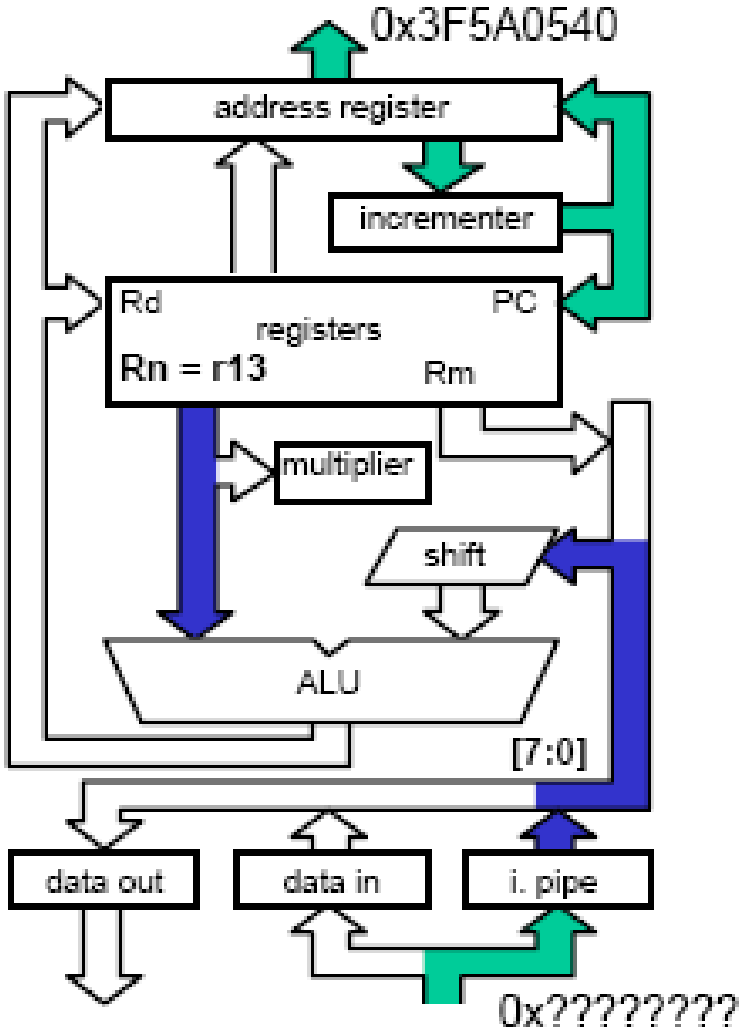


EXECUTE

Register read.

The value held in the source register, r13, (0x0F60E457) is read onto the A bus and value 0x000000E1 is read onto the B bus from the least sig. byte of the instruction.

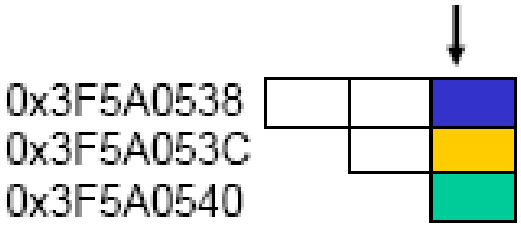


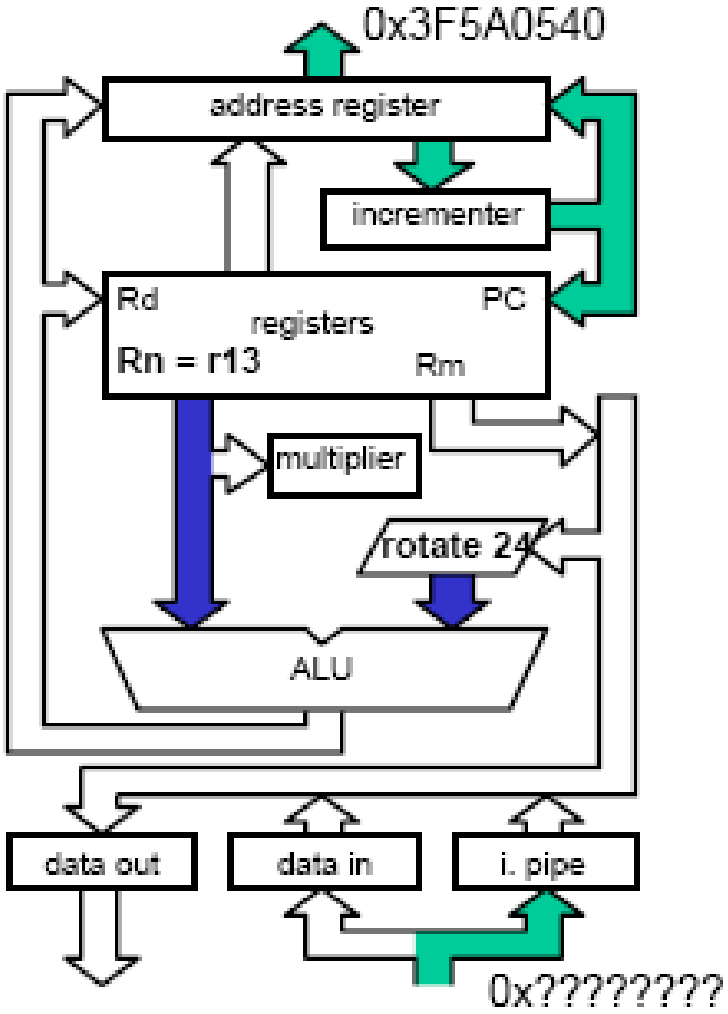


EXECUTE

Barrel shift.

The immediate value is rotated RIGHT by 24 bits (left by 8 bits) and the result, 0x0000E100, is passed to the second input of the ALU.





Explanation

The ARM7 barrel shifter only supports ‘rotate right’, not ‘rotate left’.

This is not a problem because a rotate left by *x* bits is equivalent to a rotate right by (32 – *x*) bits.

So the immediate value, #57600 = 0xE100 is equivalent to 0xE1 either rotated left by 8 bits or rotated right by 24 bits.

Hence the rotate value in the mnemonic is 12 (= 24 / 2), not 4 (= 8 / 2).

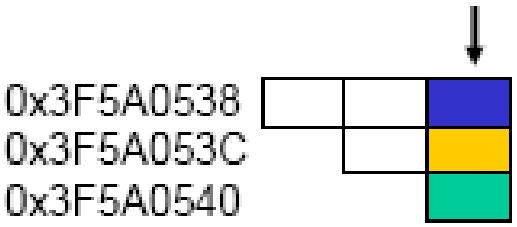
For Internal Use Only!

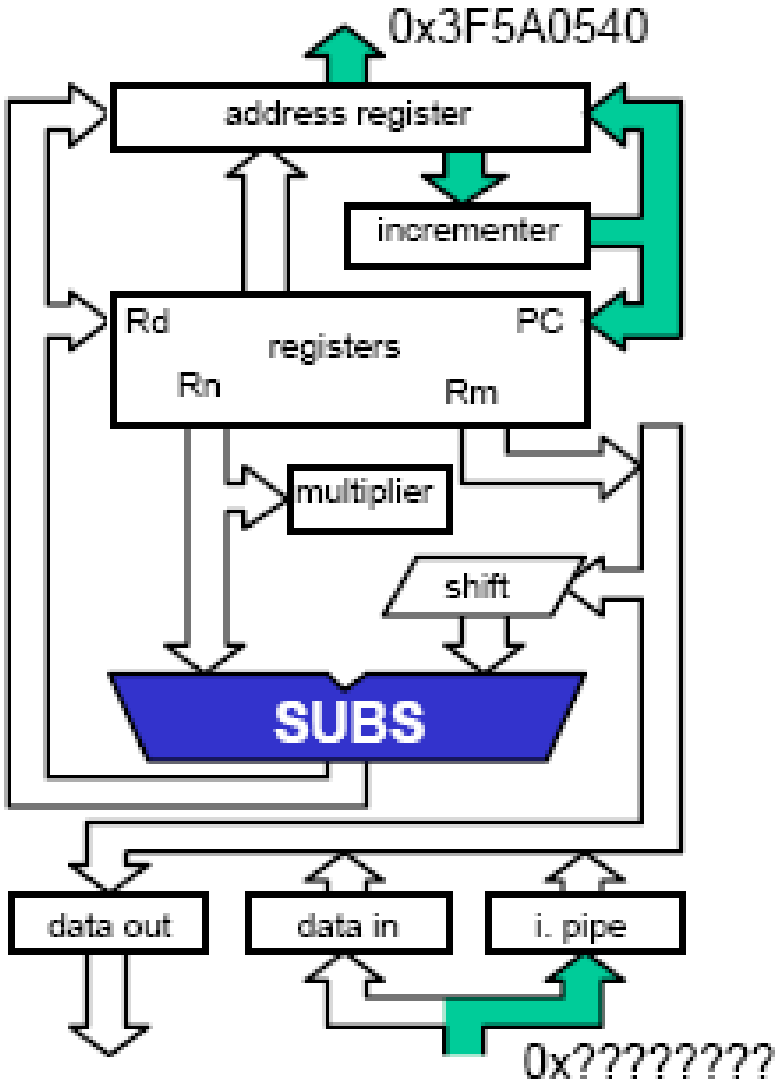
EXECUTE

ALU

The function required by the instruction, SUBS, is performed by the ALU.

The result, 0x0F61C557, is loaded onto the ALU bus.

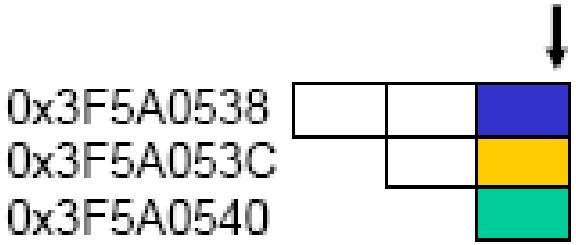


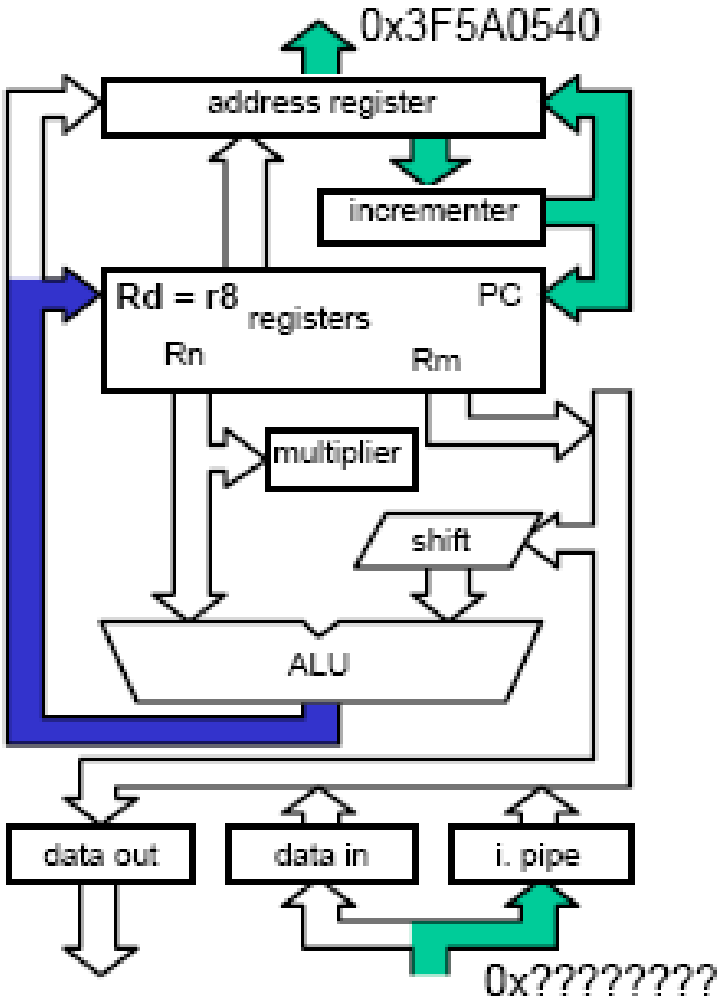


EXECUTE

Register write

The value on the ALU bus, 0x0F61C557, is written into the destination register, r8.





ARM core: not pure RISC

ARM cores do not implement all of the features proposed by Patterson and Ditzel. The features included are:

1. Load-store architecture;
2. Fixed length 32-bit instructions with few formats;
3. Hardwired combinational decode logic;
4. Pipelined execution.

The rejected features are:

1. The large register bank, the ARM only has 16 registers;
2. Delayed branches;
3. Single-cycle execution of all instructions.

For Internal Use Only!

ARM: multiple cycle instructions

There are three main ARM instruction types that are not executed in a single clock cycle:

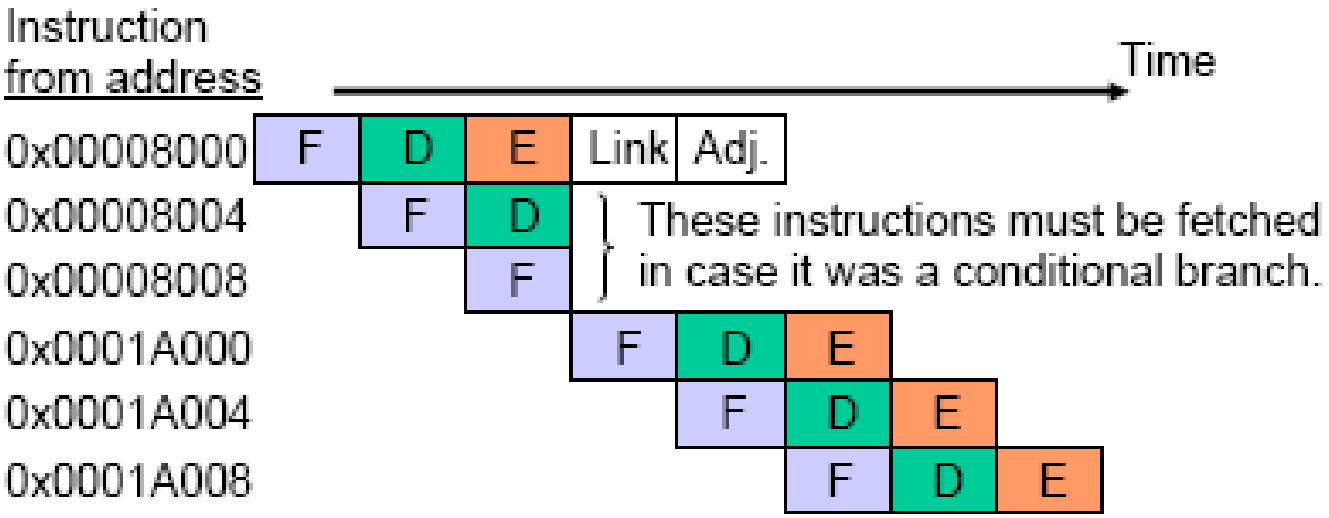
1. Multiplication – could be executed in a single cycle but would need a disproportionately large no. of gates;
2. Memory data access – there needs to be two memory accesses, one for the instruction & one for the data, so the ARM7 uses 2 cycles. Other processors use 2 buses, 2 caches or 2 memories i.e. Harvard architecture. ARM uses extra cycle for features e.g. auto-indexing.
3. Branches – result in pipeline flush. Wasted cycles used to update 'link register'.

For Internal Use Only!

ARM7 pipeline and Branch

A branch instruction will reload the program counter so that two cycles are lost .
e.g.

assume that the instruction at address 0x00008000 is Branch to 0x0001A000



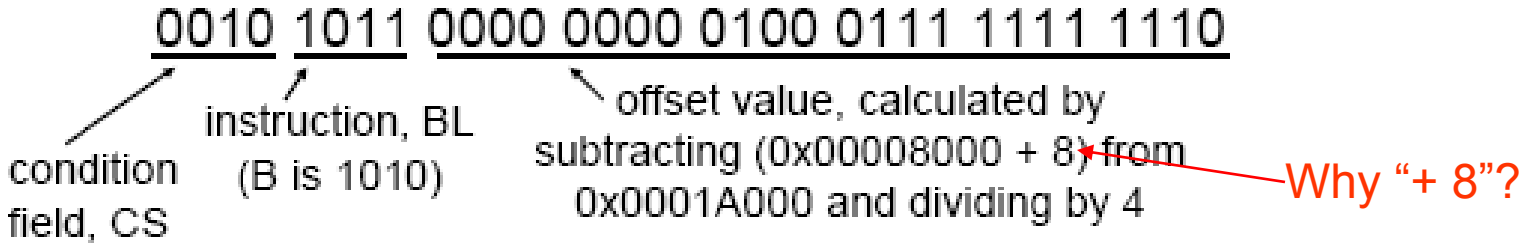
ARM7 branch example

Consider an ARM7 ‘branch and link’ instruction e.g.

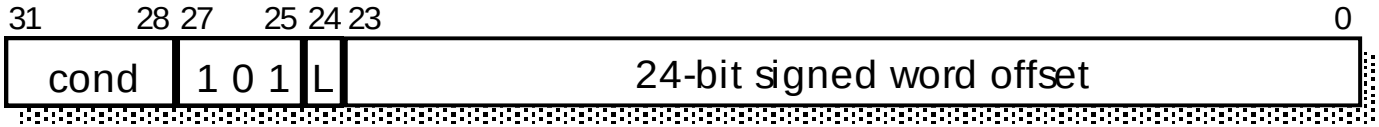
BLCS label

Branch to memory address given by label by reloading the program counter if the carry flag is set (C=1).

Assuming the memory addresses are 0x00008000 for the branch instruction and 0x0001A000 for the label, then the machine code for this instruction is 0x2B0047FE.



Branch and Branch with Link binary encoding

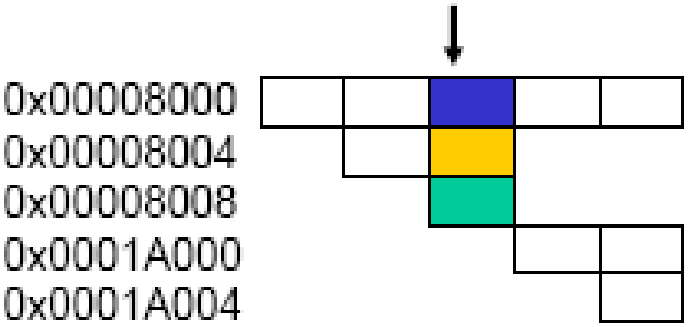


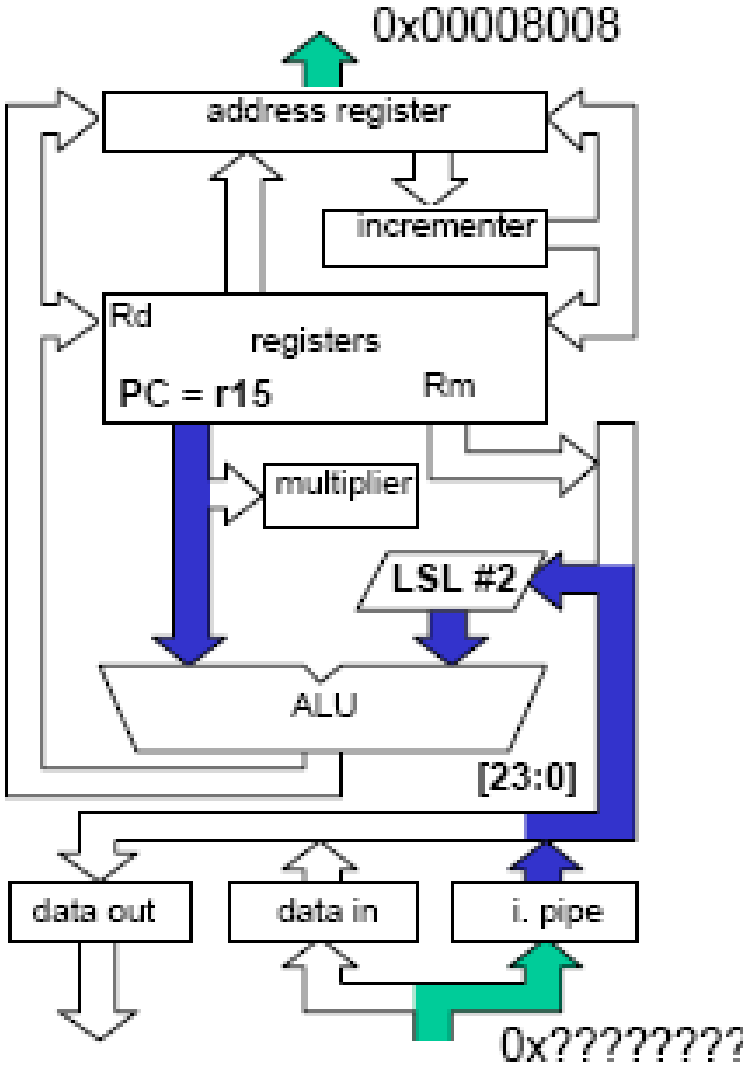
EXECUTE

First clock cycle.

Value (0x00008008) in r15 (PC) is loaded onto the A bus.

Lowest 24 bits of instruction (0x000047FE) is loaded onto B bus and left shifted 2 bits by barrel shifter (0x00011FF8).

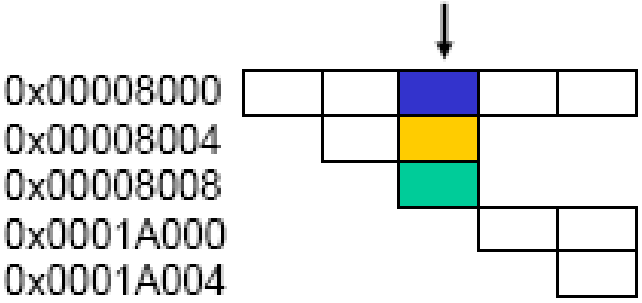


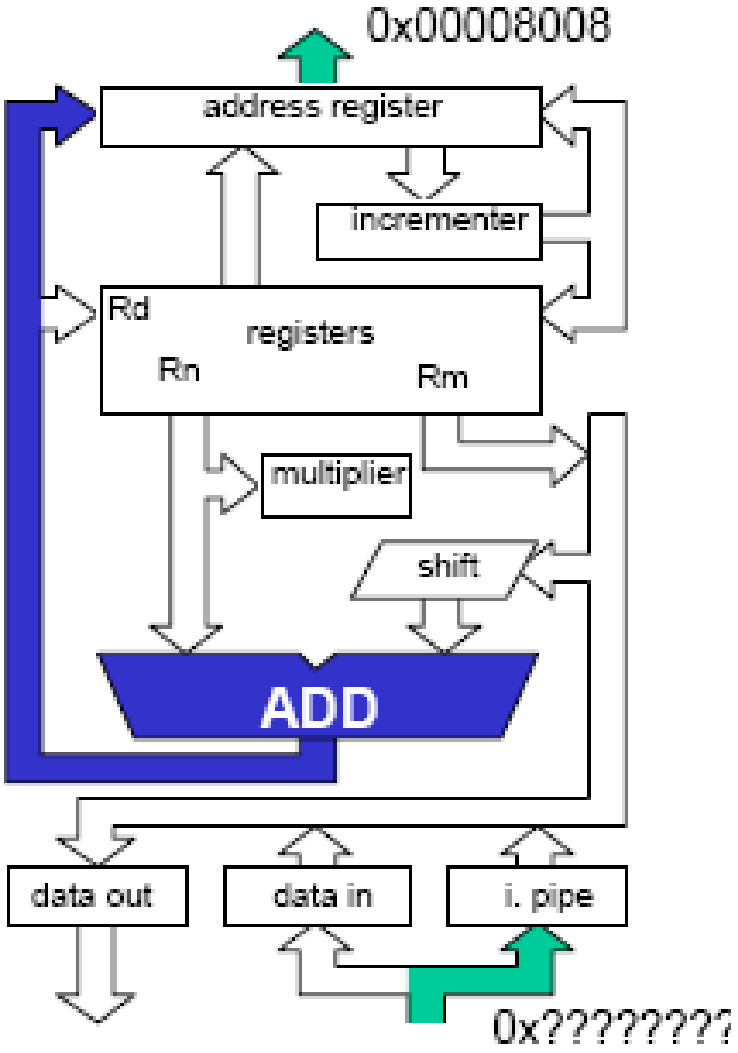


EXECUTE

First clock cycle (cont.)

ALU adds two values, (0x00008008 & 0x00011FF8) to find a new memory address (0x0001A000) to be loaded into the address register.

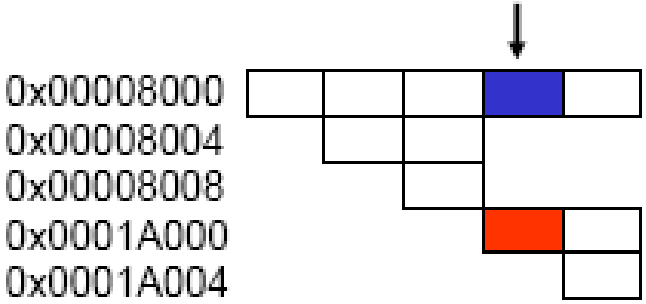


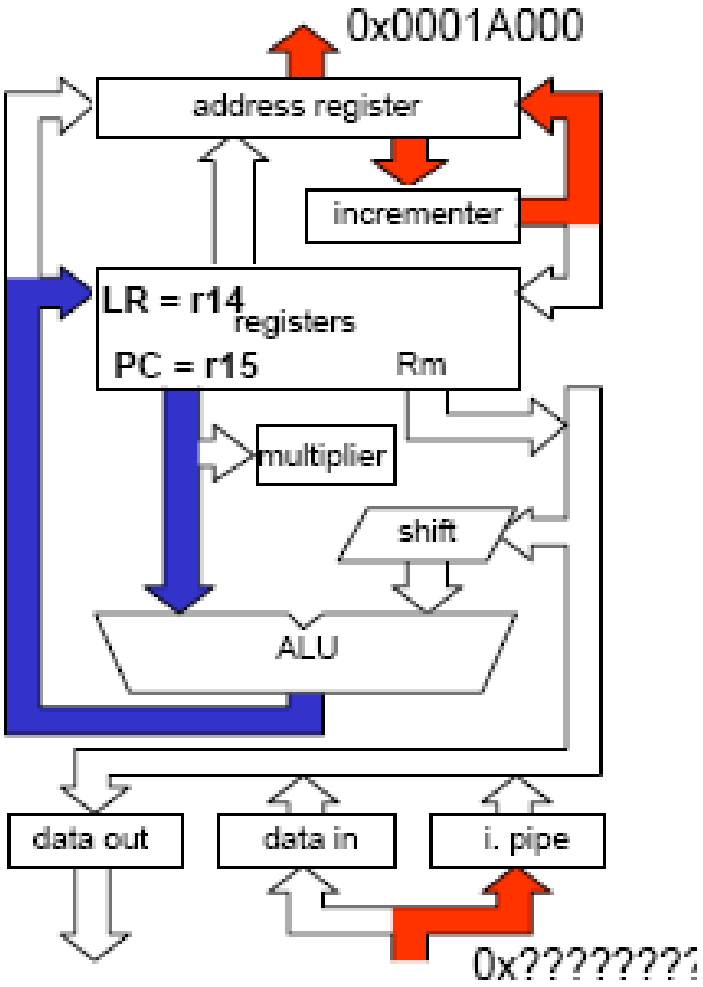


EXECUTE

Second clock cycle (link).

Value (0x00008008) in r15 (PC) is loaded onto the A bus and passed through the ALU (no action) to r14; that is the link register (LR) that holds the 'return address'.



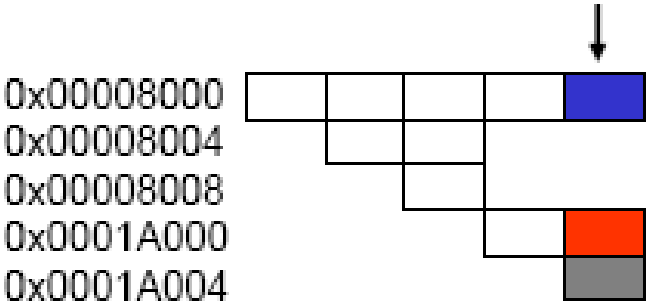


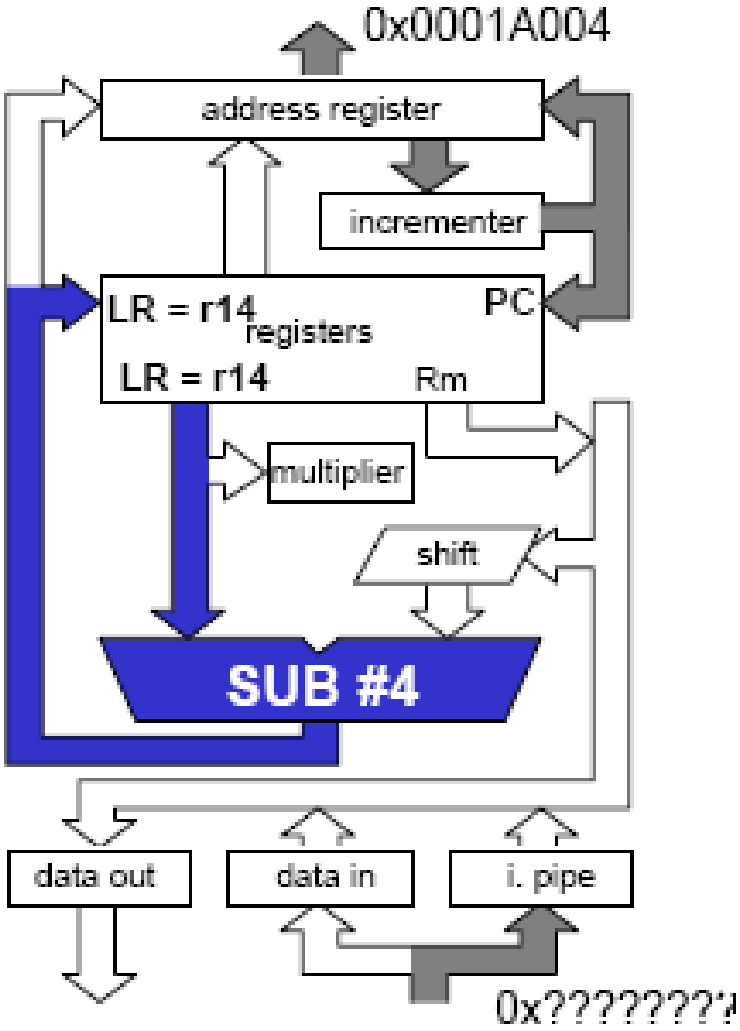
EXECUTE

Third clock cycle (adjust).

Value in r14 (LR) is adjusted by subtracting 4 in the ALU & the new value (0x0008004) is stored back in r14.

r15, the PC, is brought up-to-date.





Notes about branch (and link)

The instructions at 0x00008004 and 0x00008008 are fetched and decoded (0x00008004) in case the test for conditional execution is not met, e.g. carry flag is clear (C=0) in this case.

The branch offset (0x000047FE) is calculated by the compiler / assembler.

It is left shifted by 2 bits because ARM instructions are ‘word-aligned’ in memory.

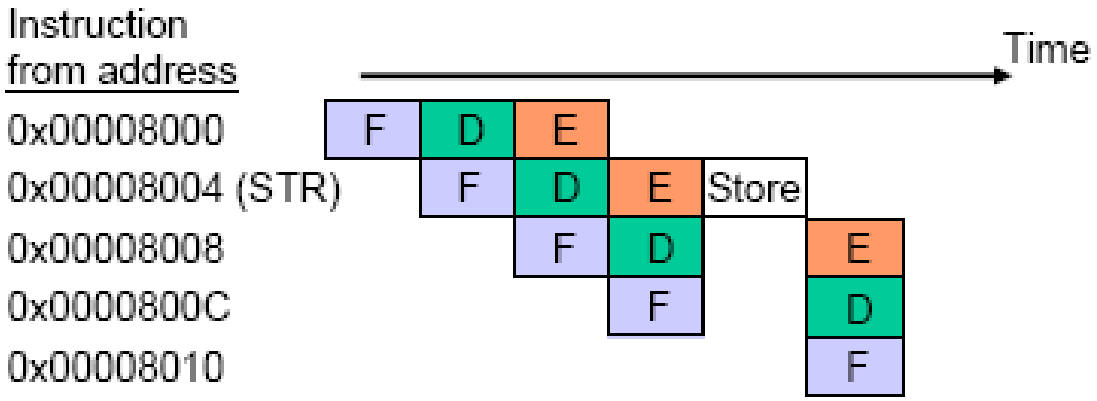
The link register must be adjusted because the program counter is 8 ahead of the instruction being executed, whereas the return address is only 4 ahead.

For Internal Use Only!

ARM7 pipeline and Store

Store instructions use the data bus to pass data from an internal CPU register to main memory.

The ARM7 has the ‘von Neuman’ architecture with a single data bus so the CPU cannot ‘pre-fetch’ an instruction at the same time as it stores data.



ARM7 Store example

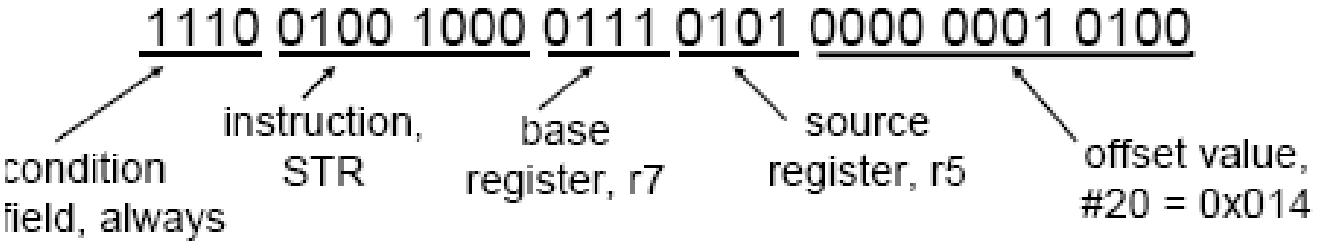
Consider a typical ARM7 Store instruction e.g.

STR r5, [r7], #20

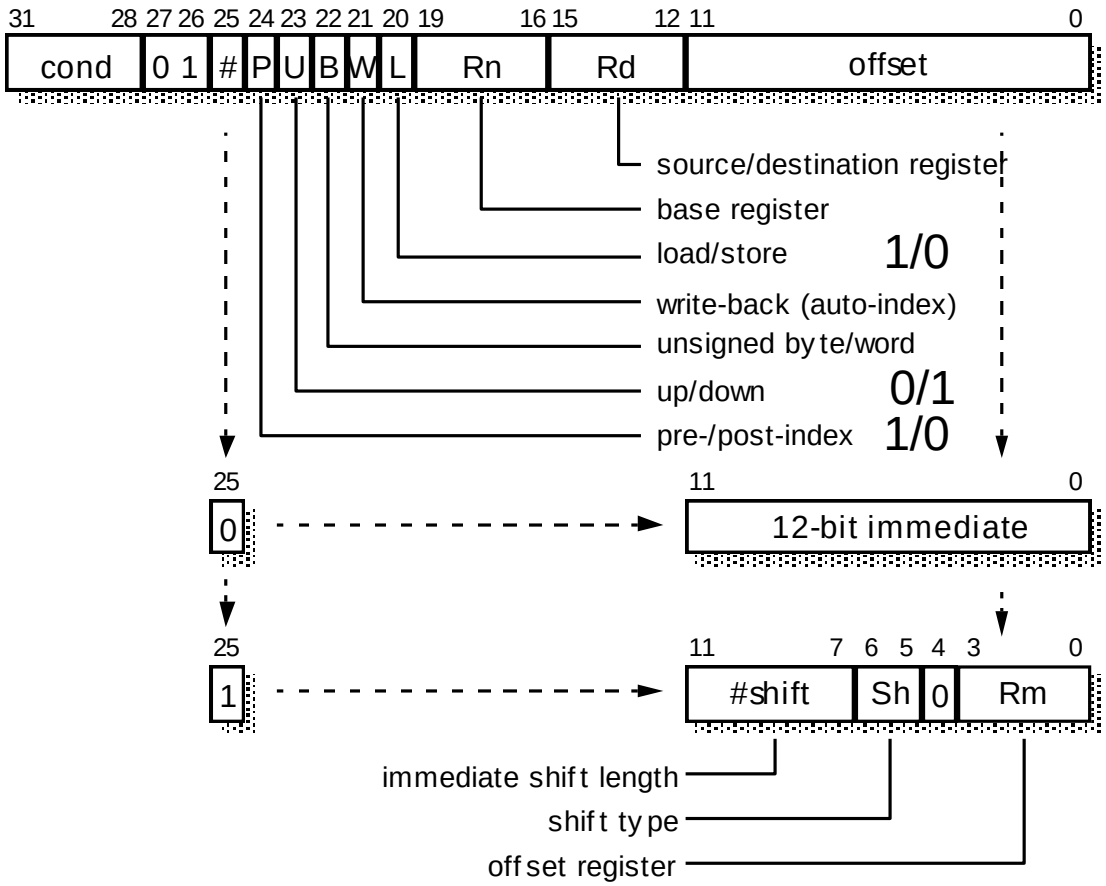
Store the contents of register r5 to a main memory location with an address given by the value in register r7.

Then ‘post-index’ the base register, r7, by adding 20 to it’s contents.

The machine code for this instruction is 0xE4875014



Single word and unsigned byte data transfer instruction binary encoding

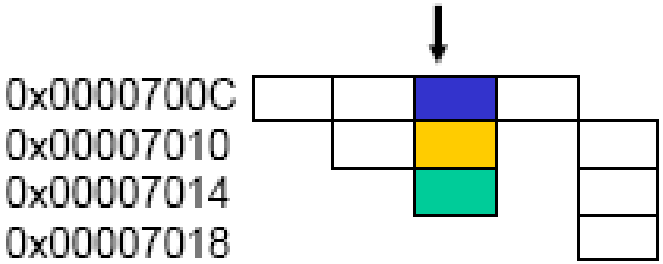


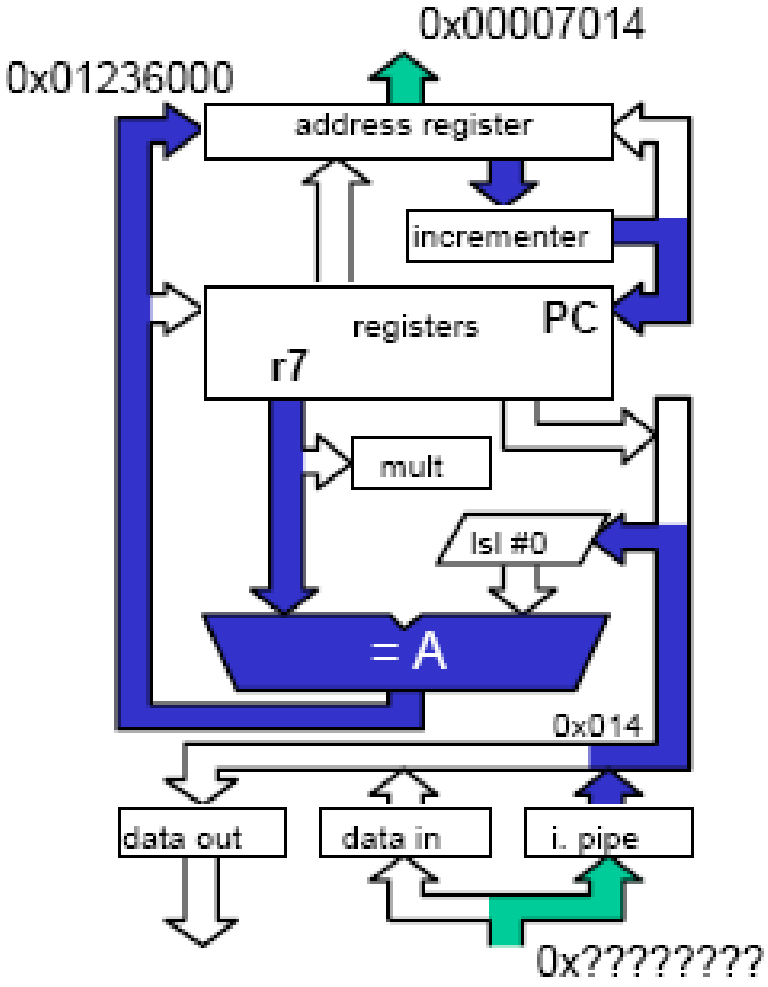
EXECUTE

First clock cycle.

Value (e.g. 0x01236000) in r7 is loaded onto the A bus and passed to the address register.

The value in the address register (e.g. 0x00007014) is incremented by 4 & stored in the PC.



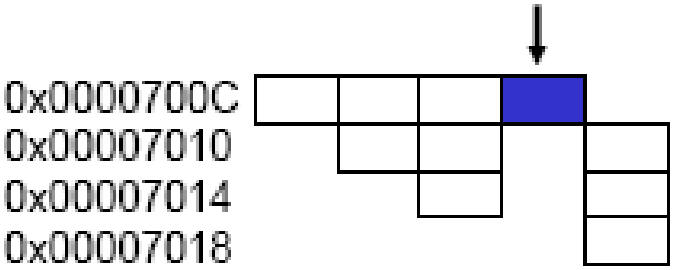


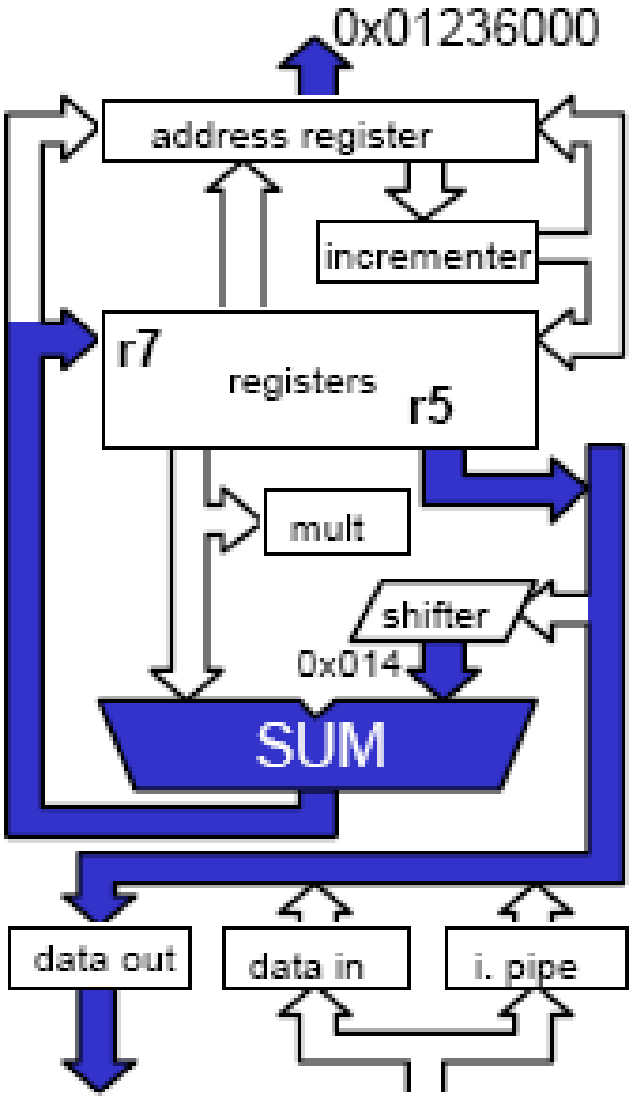
EXECUTE

Second clock cycle.

Offset value (0x014) is added to 0x01236000 and the sum, 0x01236014, is written to the base register, r7.

The value (e.g. 0x0004A000) in the source register, r5, is loaded into the data out port.

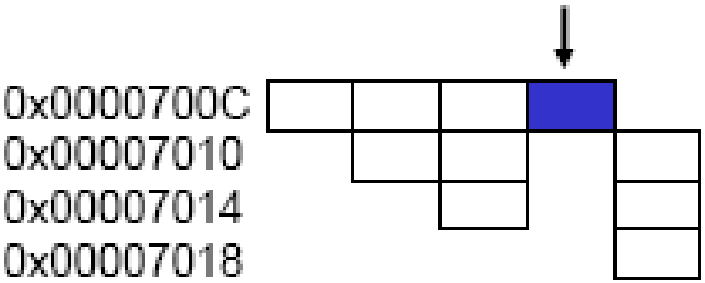


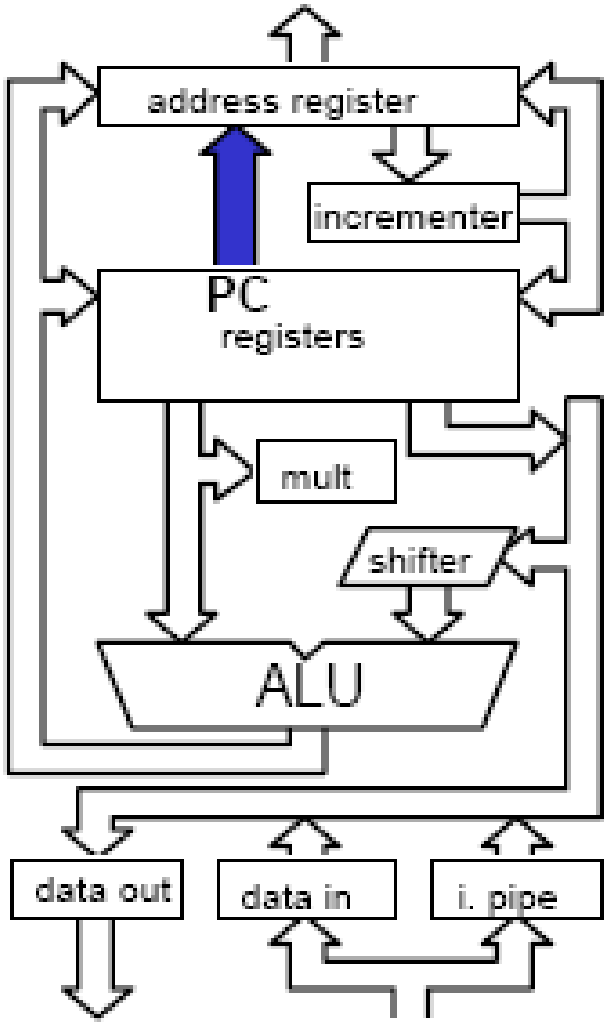


EXECUTE

Second clock cycle (cont.)

Finally the value in the program counter, 0x00007018, is loaded into the address register ready for the next instruction fetch.

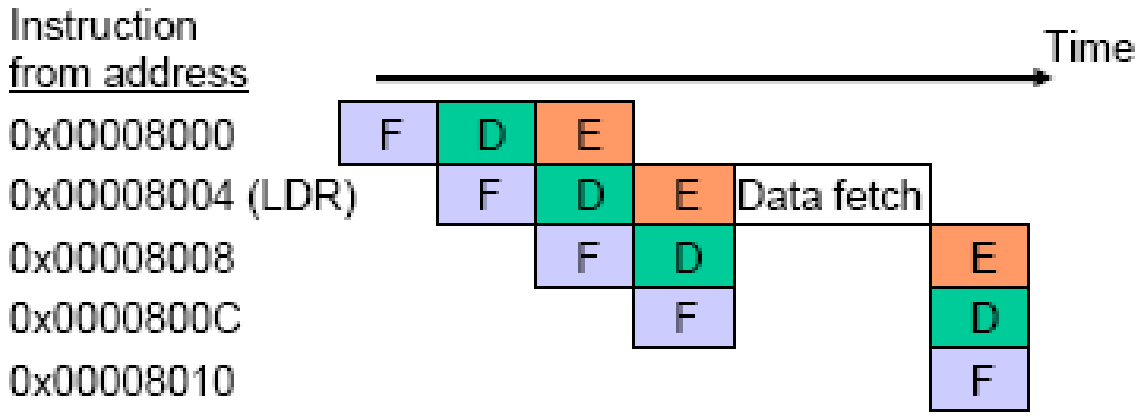




ARM7 pipeline and Load

Load instructions also use the data bus to fetch data from memory so that the data bus cannot be used to ‘pre-fetch’ an instruction at the same time.

Load instructions use a third cycle to write the data into a CPU register.



ARM7 Load example

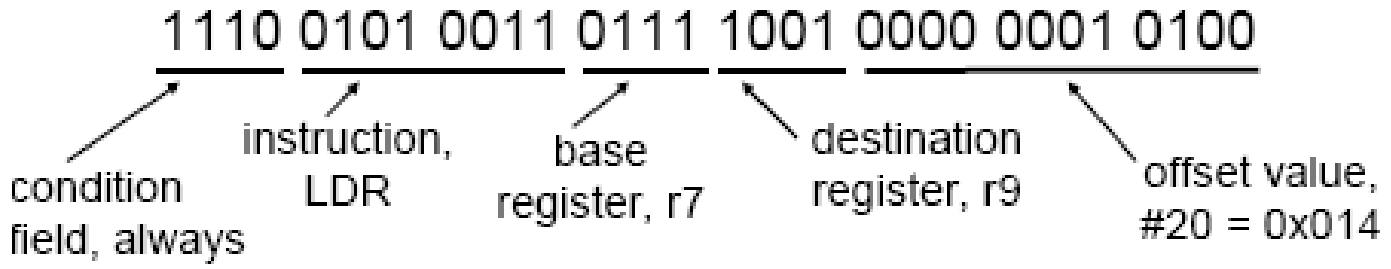
Consider a typical ARM7 Load instruction e.g.

LDR r9, [r7, #-20]!

Load register r9 from a main memory location with an address given by the value in register r7 minus 20; ‘pre-indexed’.

The base register, r7, is updated to it’s new value because the mnemonic contains the ‘pling’: !

The machine code for this instruction is 0xE5379014



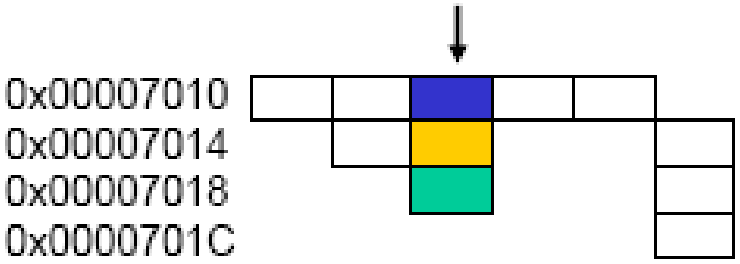
EXECUTE

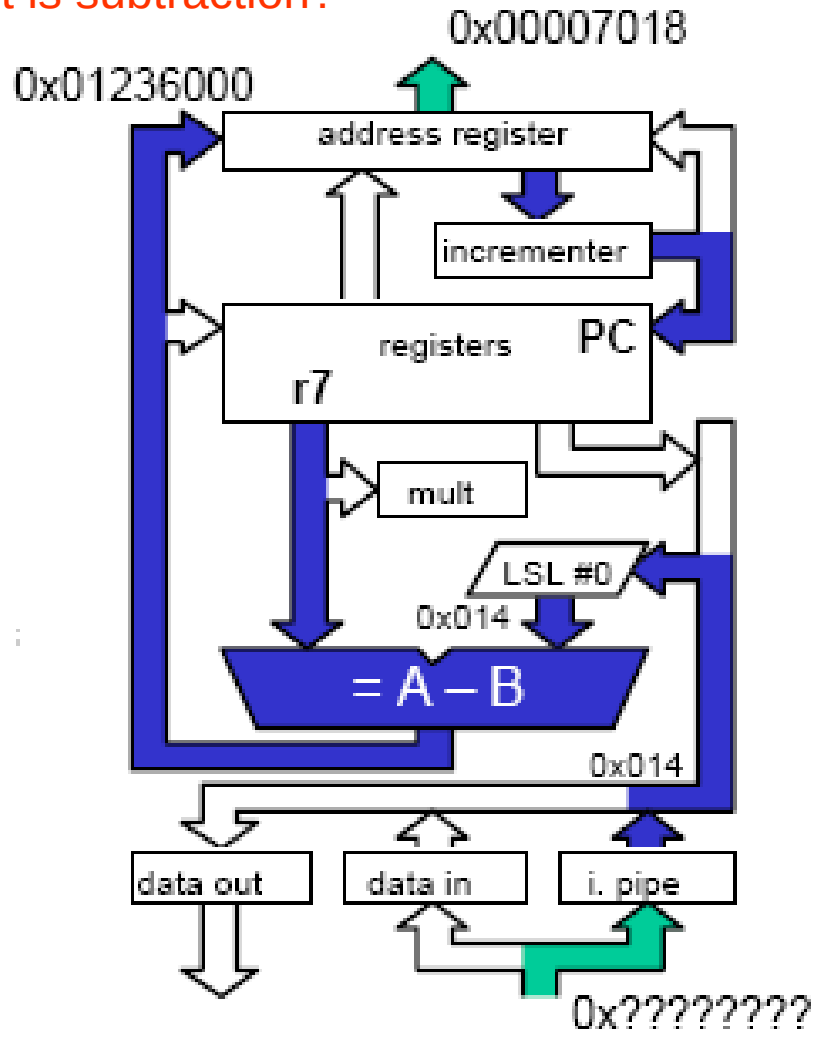
How to know it is subtraction?

First clock cycle.

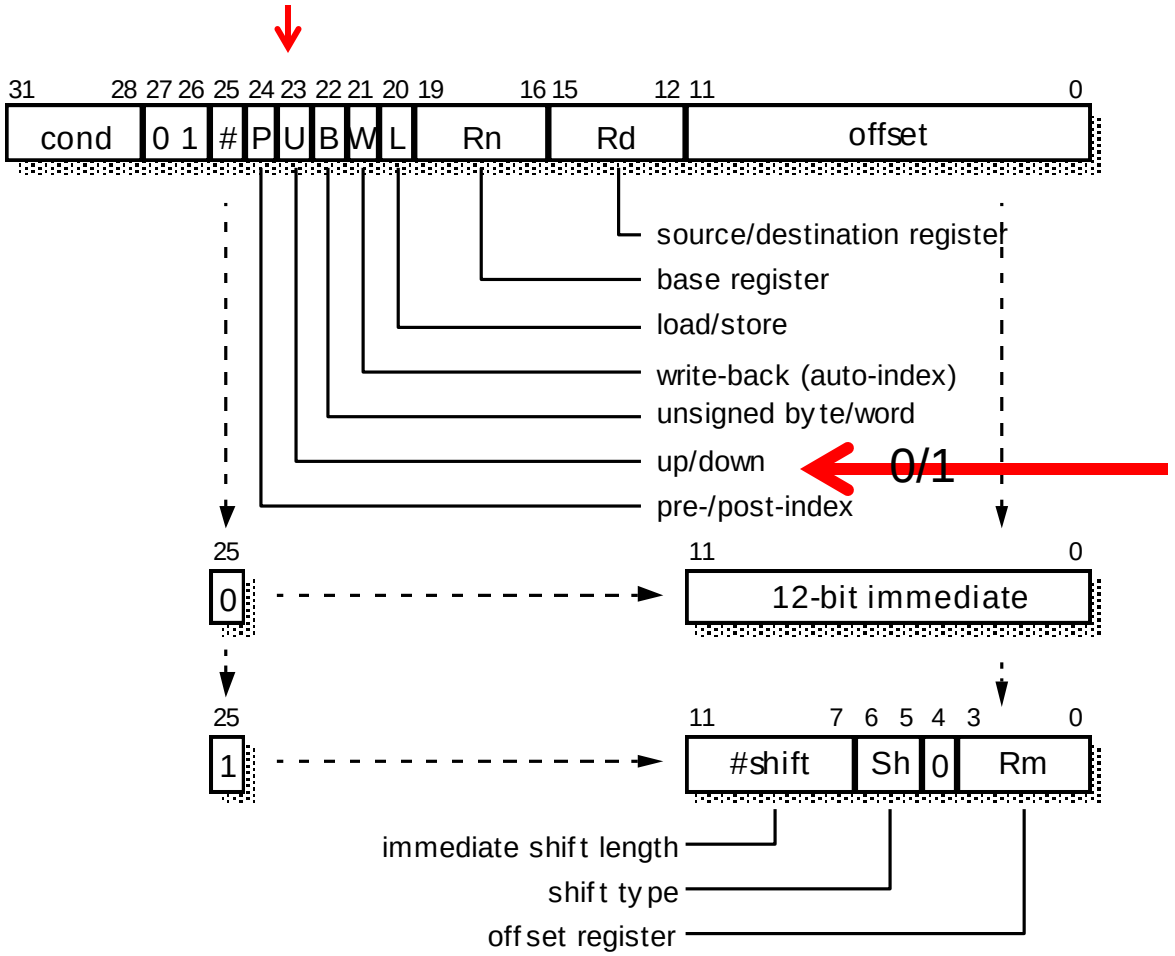
The offset, 0x014, is subtracted from the value, 0x01236014, in r7 and passed to the address register.

The value in the address register (0x00007018) is incremented by 4 & stored in the PC.





Single word and unsigned byte data transfer instruction binary encoding

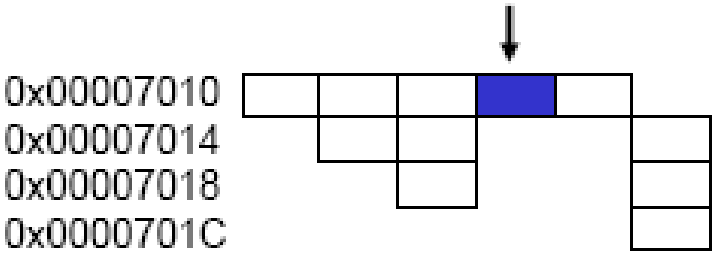


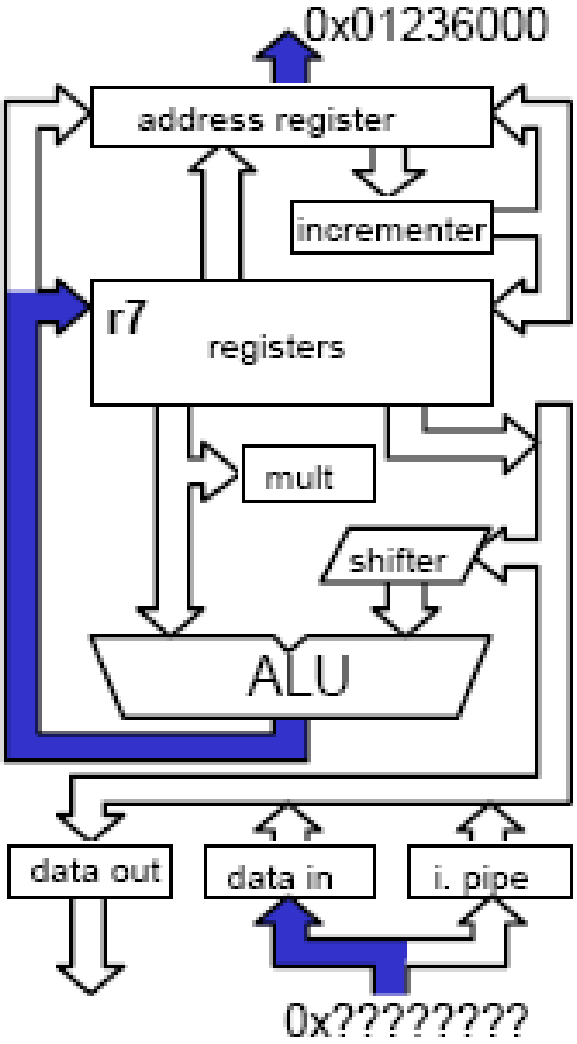
EXECUTE

Second clock cycle.

Updated address value, 0x01236000, is written to the base register, r7, if the mnemonic contains the pling.

CPU waits for data fetch from memory address 0x01236000.



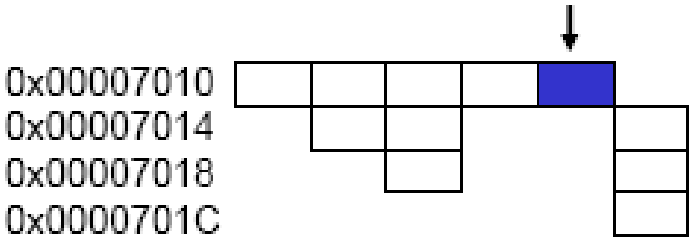


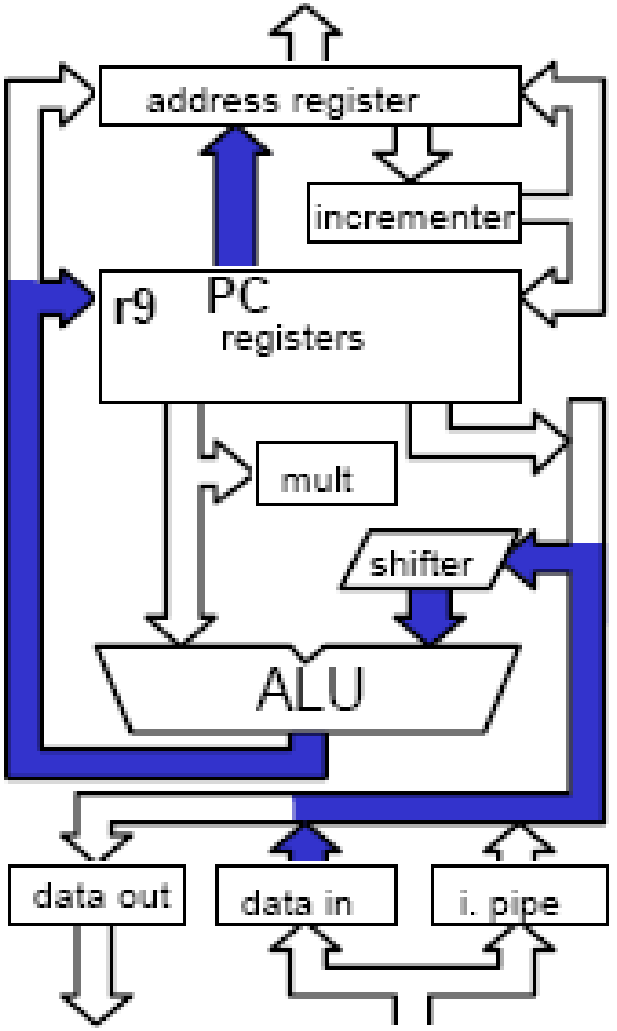
EXECUTE

Third clock cycle

Value from ‘data in’ port is loaded into the destination register, r9.

Value in the program counter, 0x0000701C, is loaded into the address register ready for the next instruction fetch.





Load/store – why so many options?

Load and store have many options e.g.

1. offset by either immediate or register,
2. positive or negative offset (or zero),
3. pre-indexing or post-indexing of base register,
4. automatic updating, ‘pling’, of base register (pre-indexing only),
5. byte or word.

All these features can be incorporated into the datapath without increasing the number of clock cycles needed to implement load (3 clocks) or store (2 clocks).

Dynamic usage of load/store

Data movement instructions, i.e. load and store, are the most commonly executed type of instruction.

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

Optional features add flexibility and saves clock cycles.

Why provide auto indexing?

Data tables are a common method of storing data.

To move through a data table, a constant value is added to (or subtracted from) the base register or ‘address pointer’.

Without auto indexing, this requires 2 instruction fetches

```
e.g.   STR r1, [r5]        ;store r1 at address in r5
       ADD r5, r5, #4      ;add 4 to base register
```

With auto indexing, only one instruction fetch is required

```
e.g.   STR r1, [r5], #4    ;post-index
```

Auto-indexing saves one clock cycle (and power dissipated in memory) by using spare resource during the load/store.

Further cycle saving

More clock cycles can be saved if more than one register is loaded or stored by a single instruction.

In the ARM microprocessor, these instructions are called ‘load multiple’ (LDM) and ‘store multiple’ (STM) and they can load/store any or all 16 registers in the register bank.

They have many uses including ‘push’ and ‘pop’ at the entry and exit from subroutines.

LDM takes (2 + n) clock cycles to execute where n is the number of registers loaded.

Likewise STM takes (1 + n) clock cycles.

So 16 LDR instructions take (3 × 16) cycles whereas LDM with n = 16 takes 18 cycles.

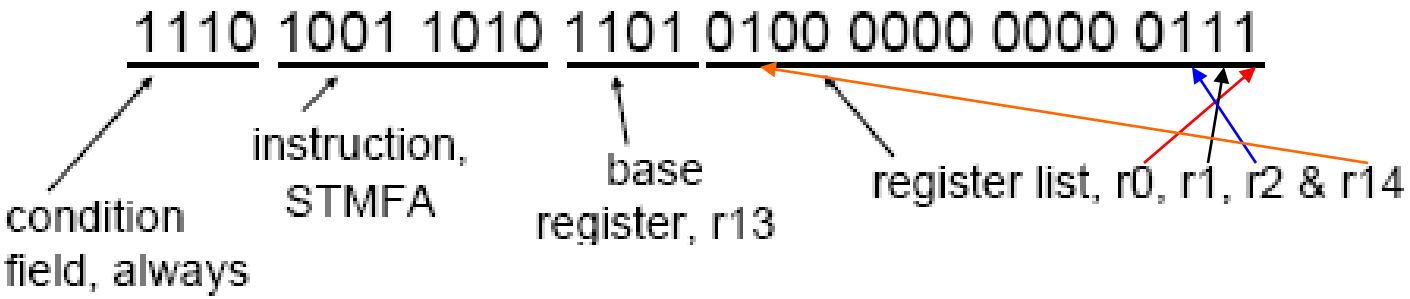
ARM7 Store Multiple example

Consider a typical ARM7 Store Multiple (push) instruction, e.g.

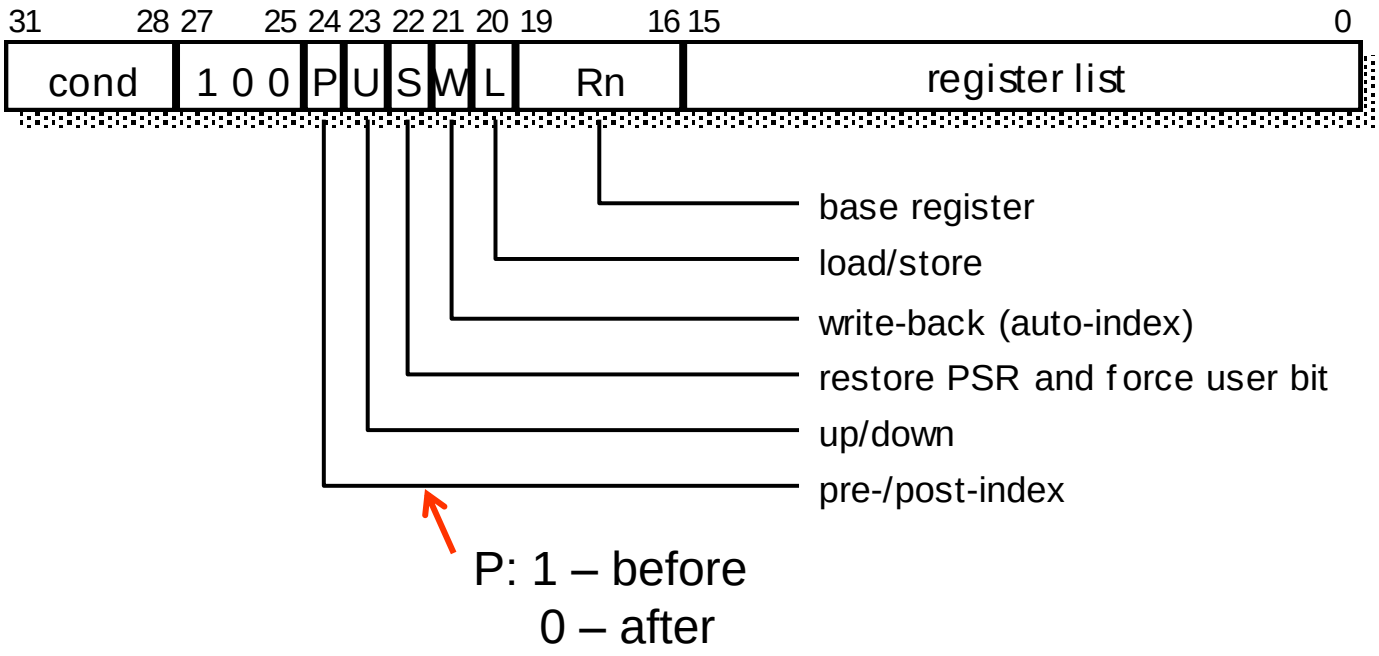
STMFA r13!, {r0-r2, r14}

Store the contents of r0, r1, r2 and r14 to a main memory stack with an address given by the value in register r13. (Push r0, r1, r2 and link register onto the stack.)

The machine code for this instruction is 0xE9AD4007



Multiple register data transfer instruction binary encoding



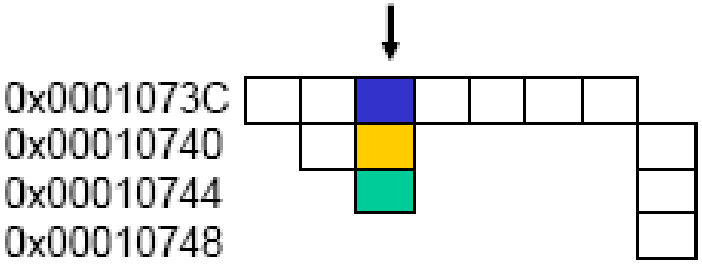
EXECUTE

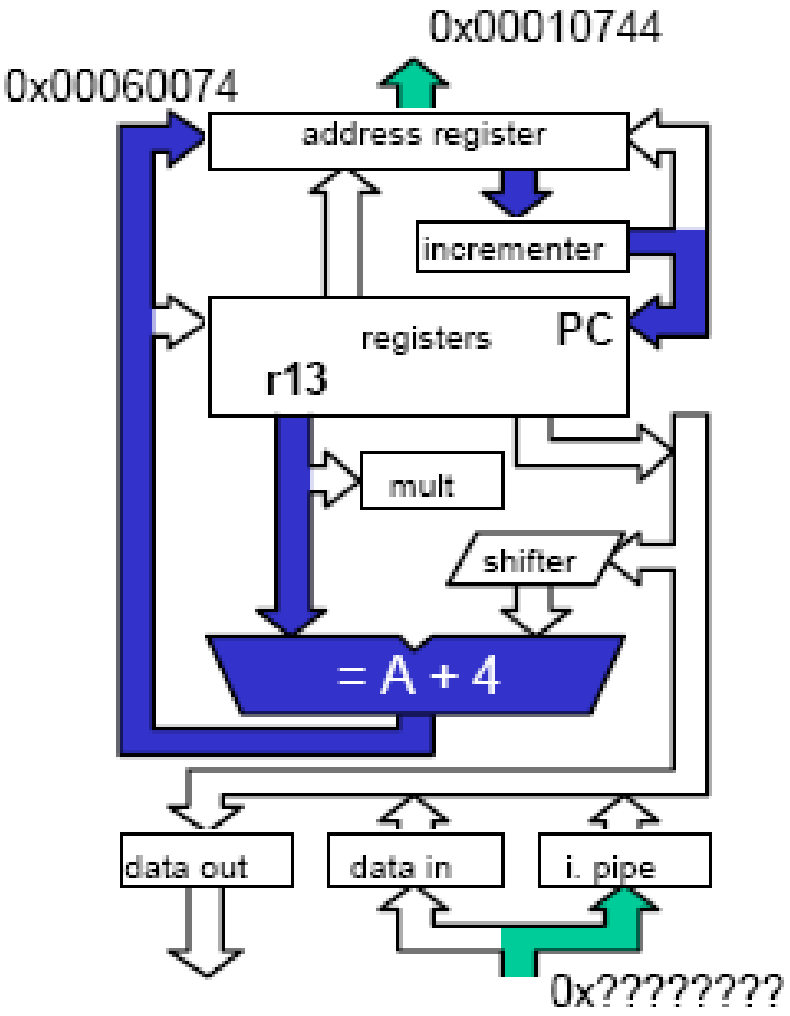
First clock cycle.

The value in the address register (e.g. 0x00010744) is incremented by 4 & stored in the PC.

P = 1

4 added to the value (e.g. 0x00060070) in r13 and passed to the address register.



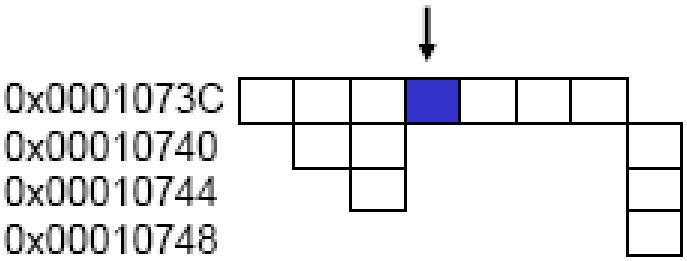


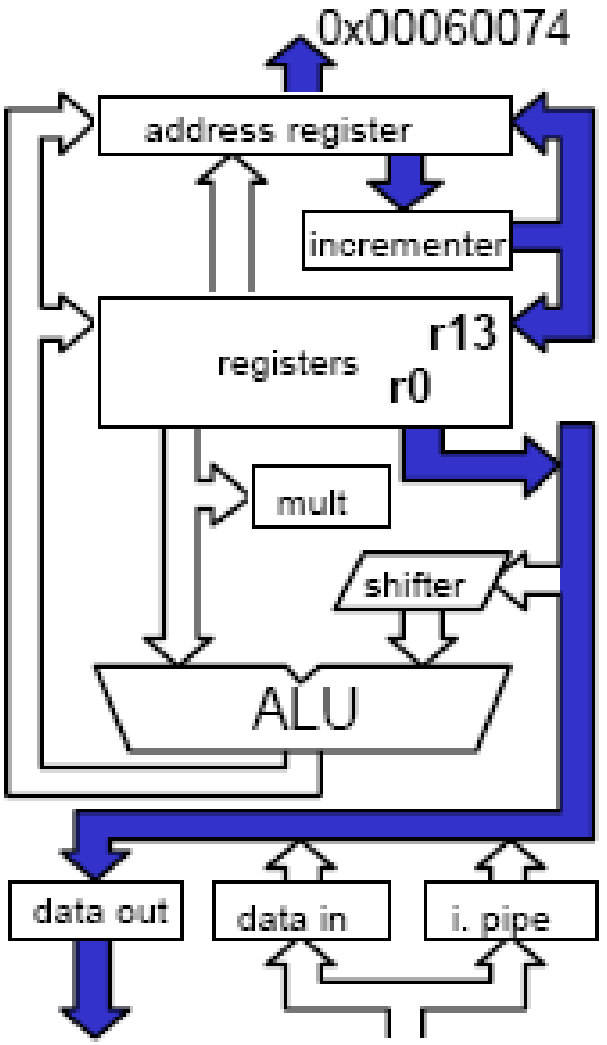
EXECUTE

Second clock cycle.

The value in the first source register, r14, is loaded into the ‘data out’ port.

The value in the address register is incremented by 4 and copied back to the base register, r13.



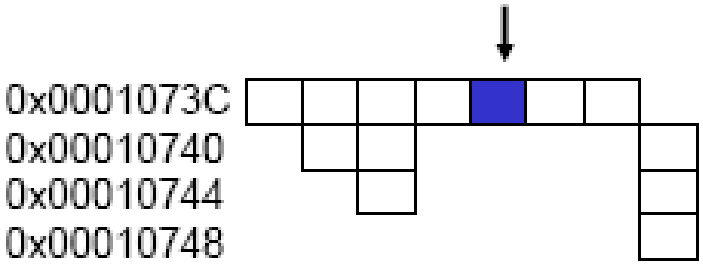


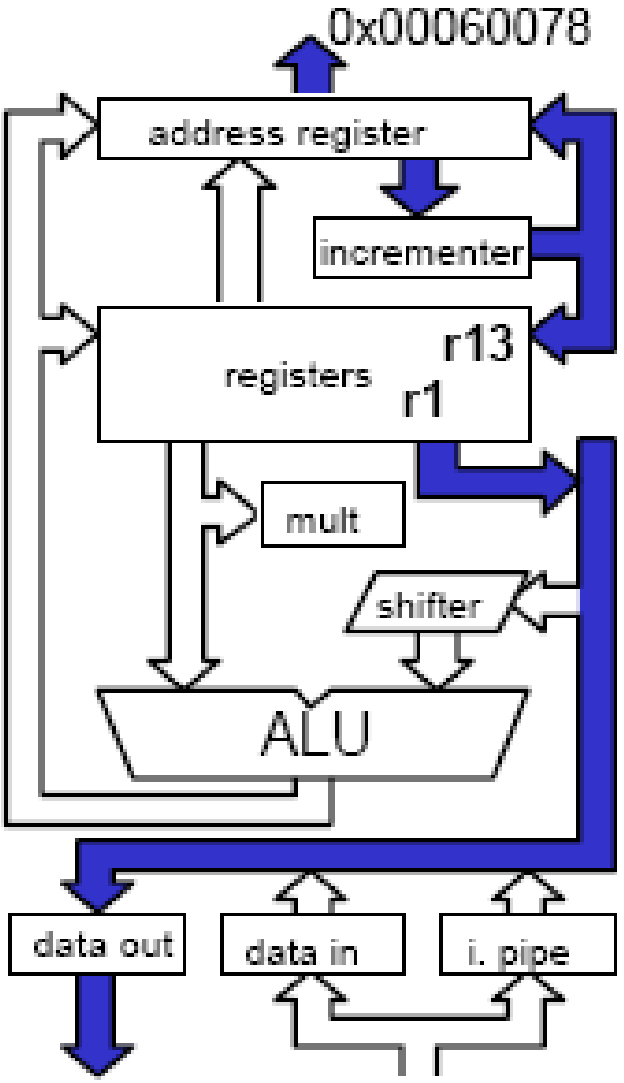
EXECUTE

Third clock cycle.

The value in the second source register, r2, is loaded into the ‘data out’ port.

The value in the address register is incremented by 4 and copied back to the base register, r13.



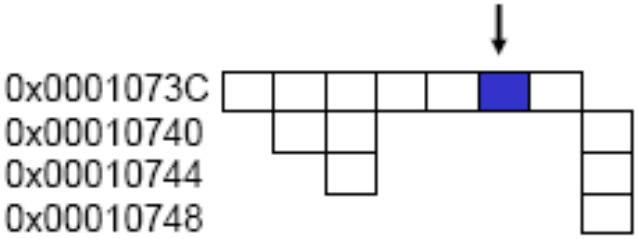


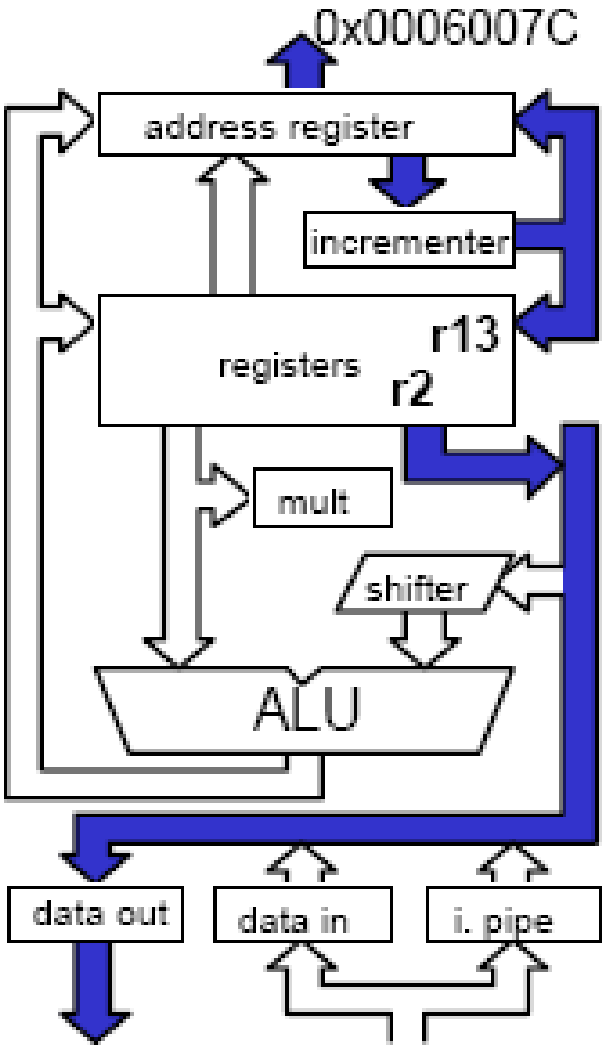
EXECUTE

Fourth clock cycle.

The value in the third source register, r1, is loaded into the ‘data out’ port.

The value in the address register is incremented by 4 and copied back to the base register, r13.



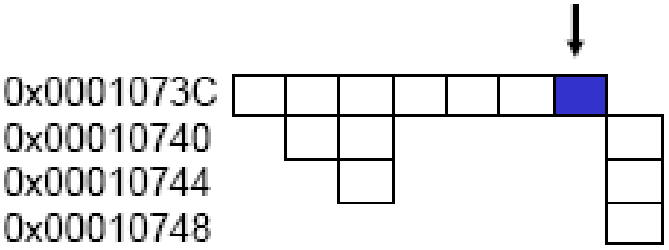


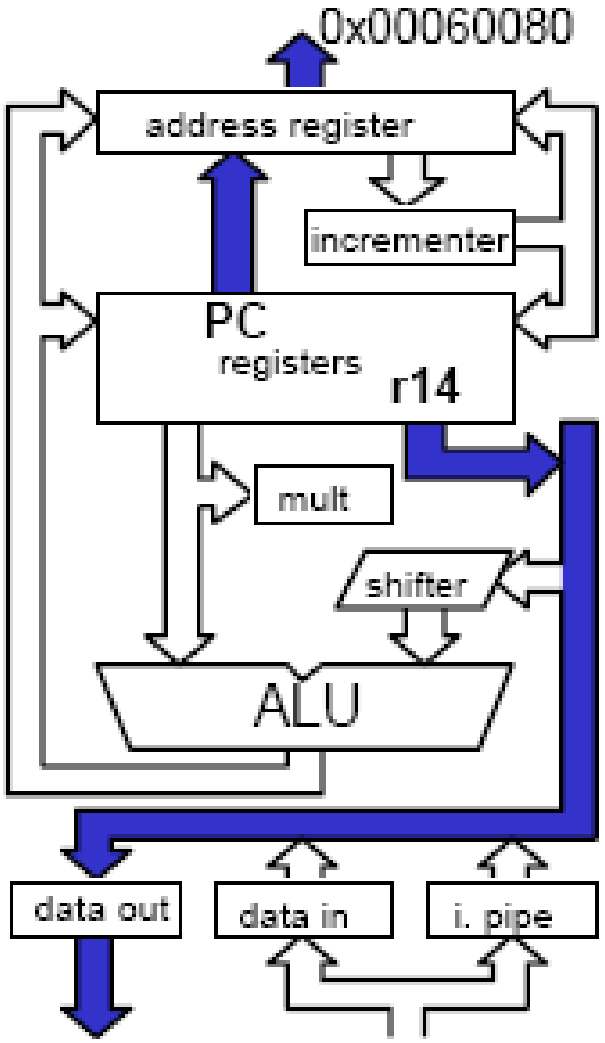
EXECUTE

Fifth clock cycle.

The value in the last register, r0, is loaded into the ‘data out’ port.

The value in PC, 0x00010748, is loaded into the address register, ready for the next instruction ‘pre-fetch’.





ARM7 performance

The ARM7 does not achieve the goal for RISC microprocessor of one instruction per clock cycle.

The ARM7 has a performance of about 1.9 CPI (CPI = clock per instruction).

However the ARM7 does make efficient use of the data bus with nearly every clock cycle used for either an instruction pre-fetch or a data load/store.

The data bus is only idle during

- the last cycle of load (LDR/LDM)
- all but the first cycle of a multiply (MUL/MLA) instruction.

Performance improvements

The data path / pipeline described so far relates to the ARM7 microprocessor only.

How can the performance be improved in the next generation (ARM9) and later versions of the ARM microprocessor?

Performance improvements can be gained by including:

- More pipelining i.e. longer pipeline
- Harvard architecture
- Delayed branches
- Additional specialized instructions

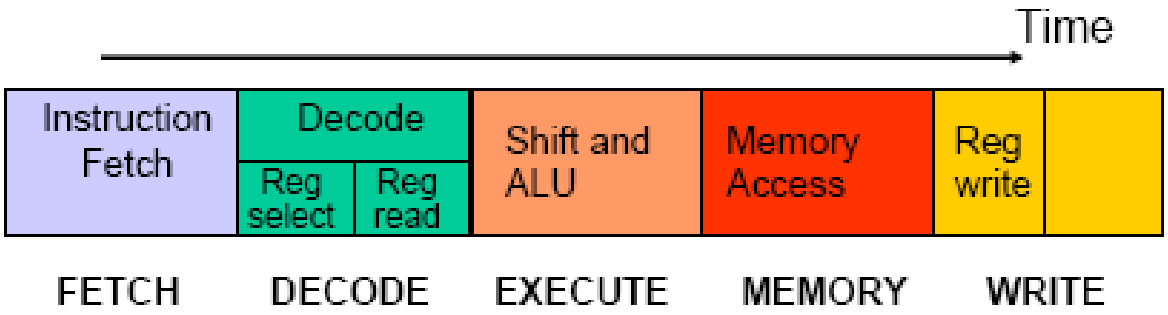
Longer pipelines

By introducing more pipeline stages, the activity in any one stage is reduced so that the clock frequency can be increased.

The bottleneck in the ARM7 3 stage (fetch, decode, execute) pipeline is the execute stage.

The ARM9 5 stage pipeline is more balanced because the ‘execute’ functions are spread over two and a half pipeline stages called execute, memory and write.

ARM9 5 stage pipeline: detail

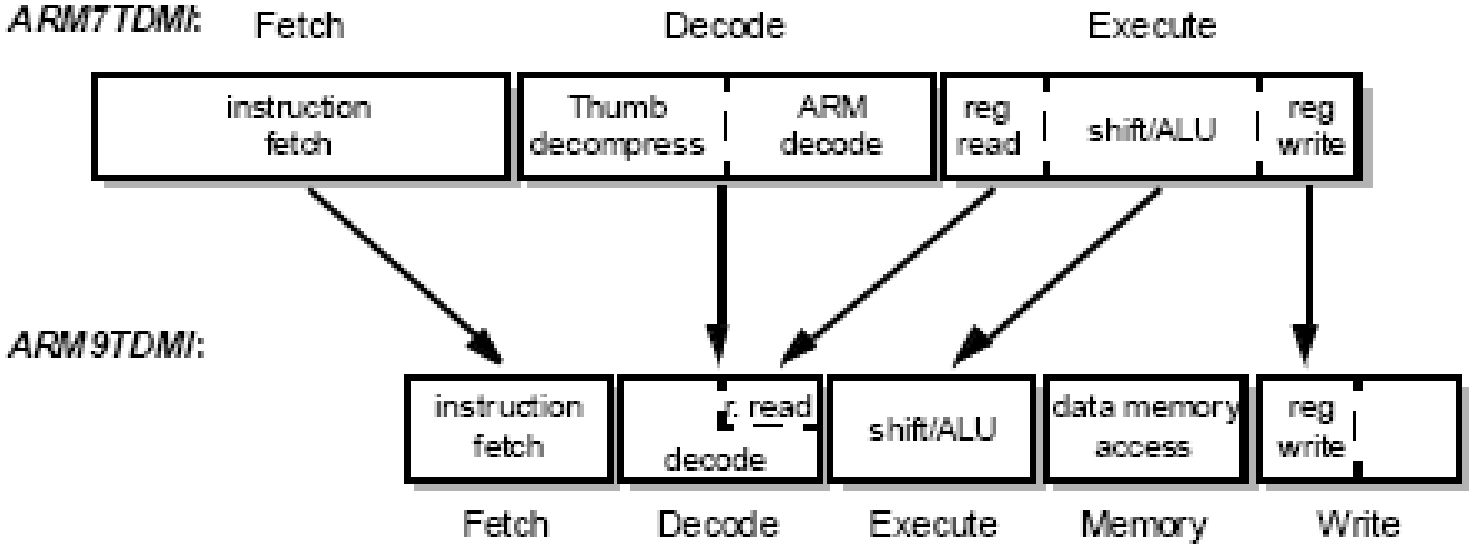


In the ARM9 pipeline both ARM and Thumb decode happen in parallel.

Register read is moved into the decode stage and register write is delayed until after a separate stage for any memory data access (load/store).

ARM7 & ARM9 pipeline comparisons

More stages = less activity per stage
= higher clock frequency
= higher performance

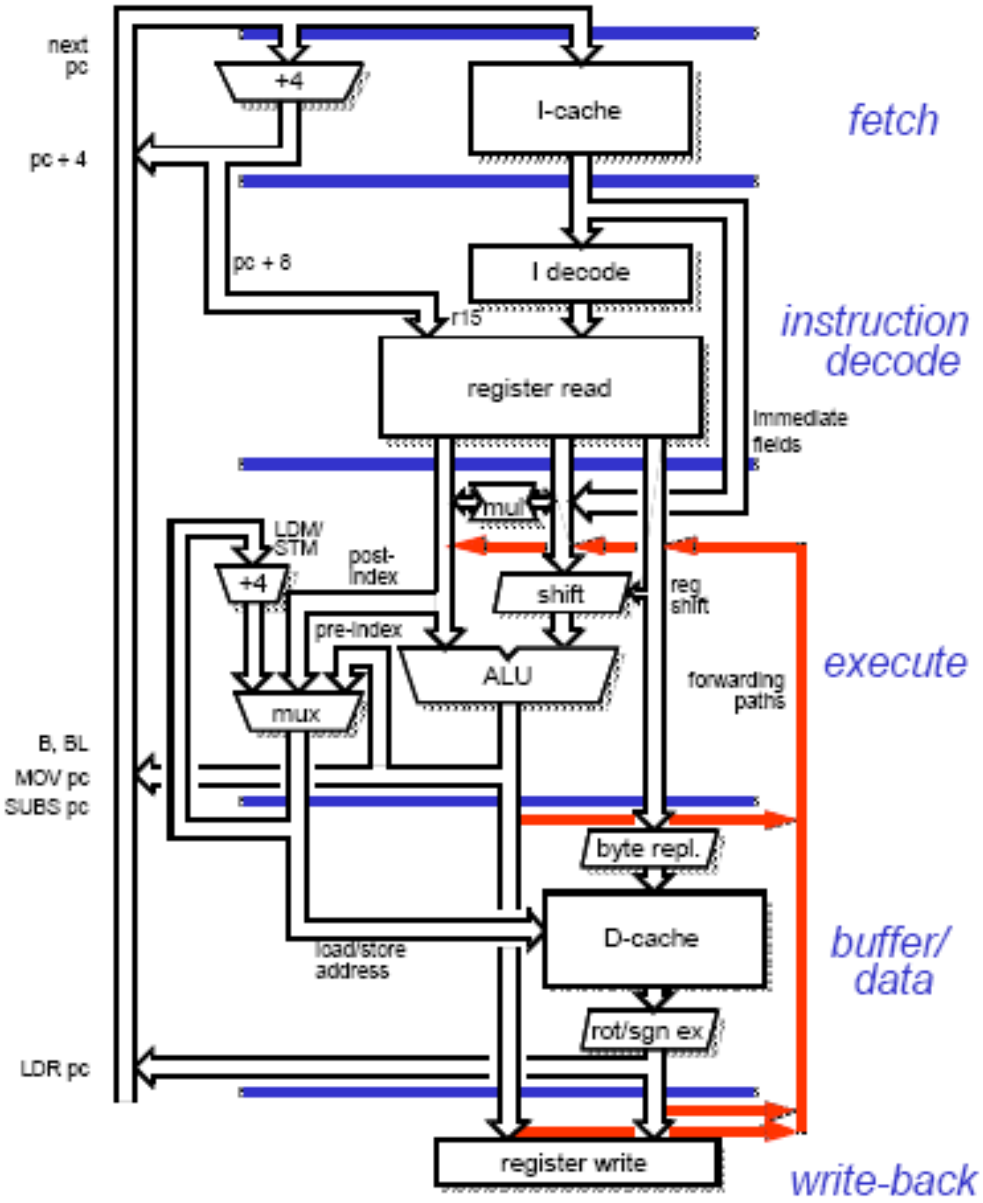


ARM9TDMI 5 stage pipeline organization

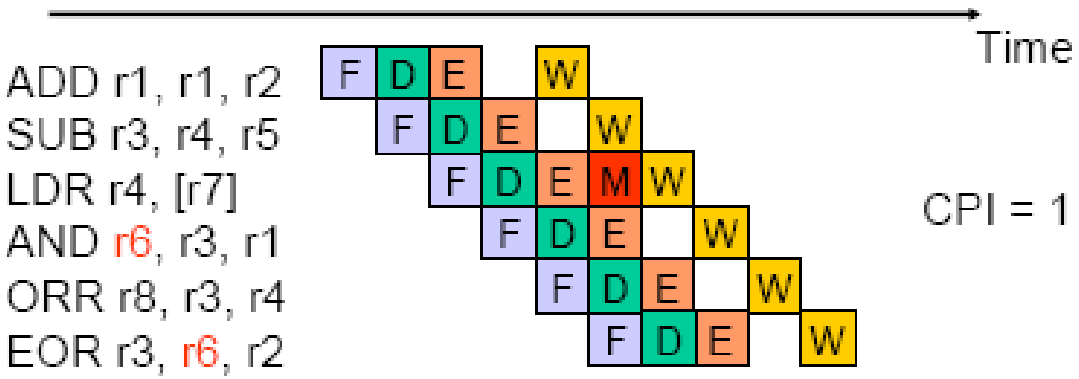
Note that in the ARM9TDMI 5 stage pipeline, a register is read in stage 2 (decode) and written in stage 5 (writeback).

This is a problem if an instruction uses a register that is changed by a previous instruction.

This problem is overcome, to some extent, by using the 'data forwarding' paths shown in red.



ARM9 optimal pipelining

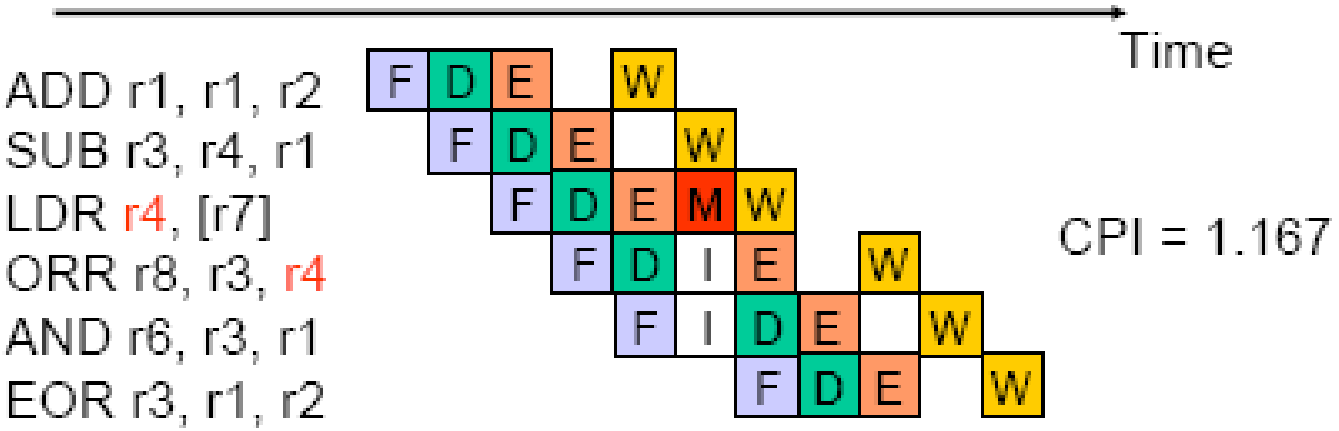


In this example only the load, LDR, uses the ‘memory’ stage.

Data can be loaded at the same time as prefetching an instruction (EOR) because the ARM9 has a data bus and an instruction bus (Harvard architecture).

Also ‘data forwarding’ allows r6 to be used before it is written.

ARM9 pipeline interlock

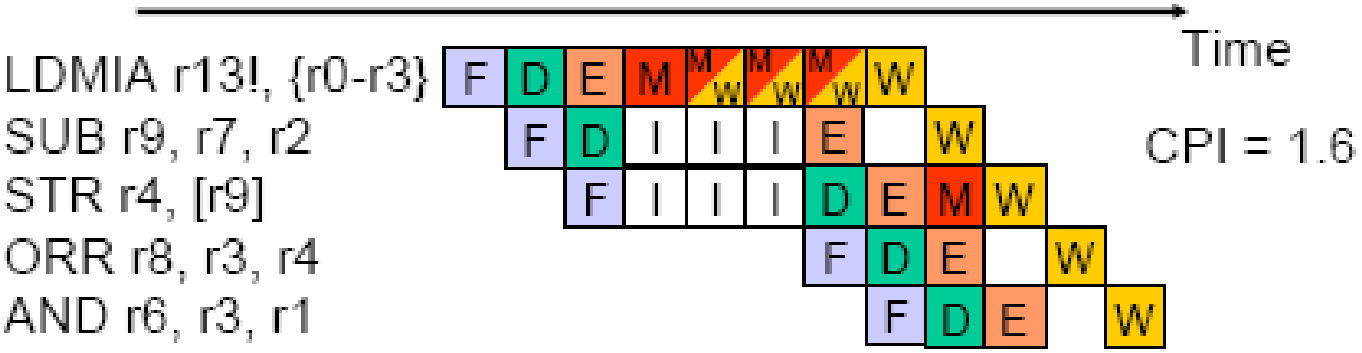


An ‘interlock’ occurs if an instruction immediately following a load uses the same register, i.e. r4 in this example.

This is called ‘operand dependency’. These 6 instructions take 7 clock cycles to execute; hence CPI = 1.167.

A good compiler should swap instructions to avoid interlocks.

ARM9 load multiple interlock



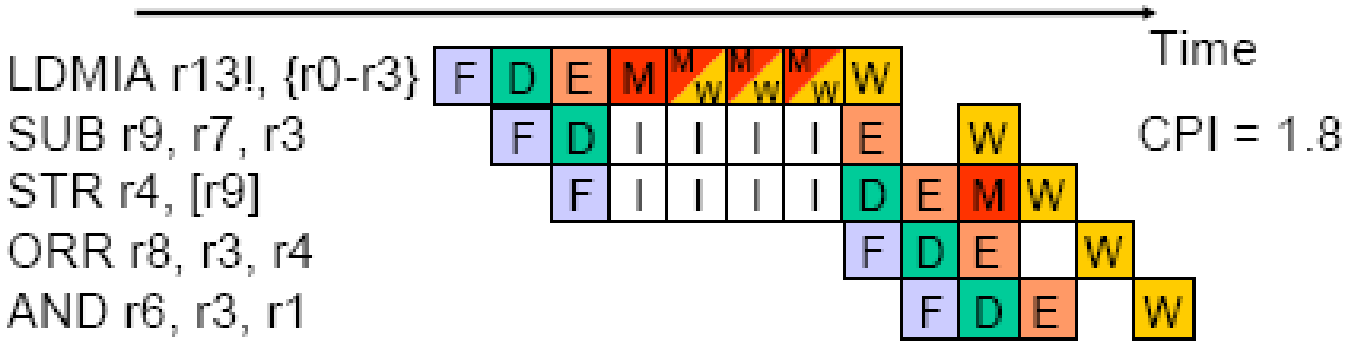
Load multiple instructions use a simultaneous memory & 'writeback' stage;

in example the loaded value is written to r0 at the same time as memory read occurs for r1.

The SUB instruction must wait for the r2 value to be loaded from memory so that 'interlock' occurs;

hence 5 instructions take 8 clock cycles.

ARM9 load multiple interlock



In ‘load multiple’, the highest specified register in the list is loaded last

so, in this example, the SUB instruction must wait for r3 to be loaded.

This adds a further interlock cycle so that the 5 instructions take 9 clock cycles to execute; CPI = 1.8.

This problem is known as ‘operand dependency’.

ARM7 v. ARM9 performance

The advantage of Harvard over von Neumann is that there are fewer lost clock cycles due to bus conflicts.

The drawback is greater complexity.

The ARM7 has an average CPI of 1.9 whereas the ARM9 has an average CPI of 1.5.

Note that this depends upon the program being executed but a typical program makes a lot of use of load and store.

The ARM9 also has a higher clock frequency because there is no bottleneck in the execute stage of the pipeline as there is with the ARM7.

ARM10: 6 stage pipeline