ARM CPU Datapath

ENCE361: Design & Architecture: Lecture Block 3

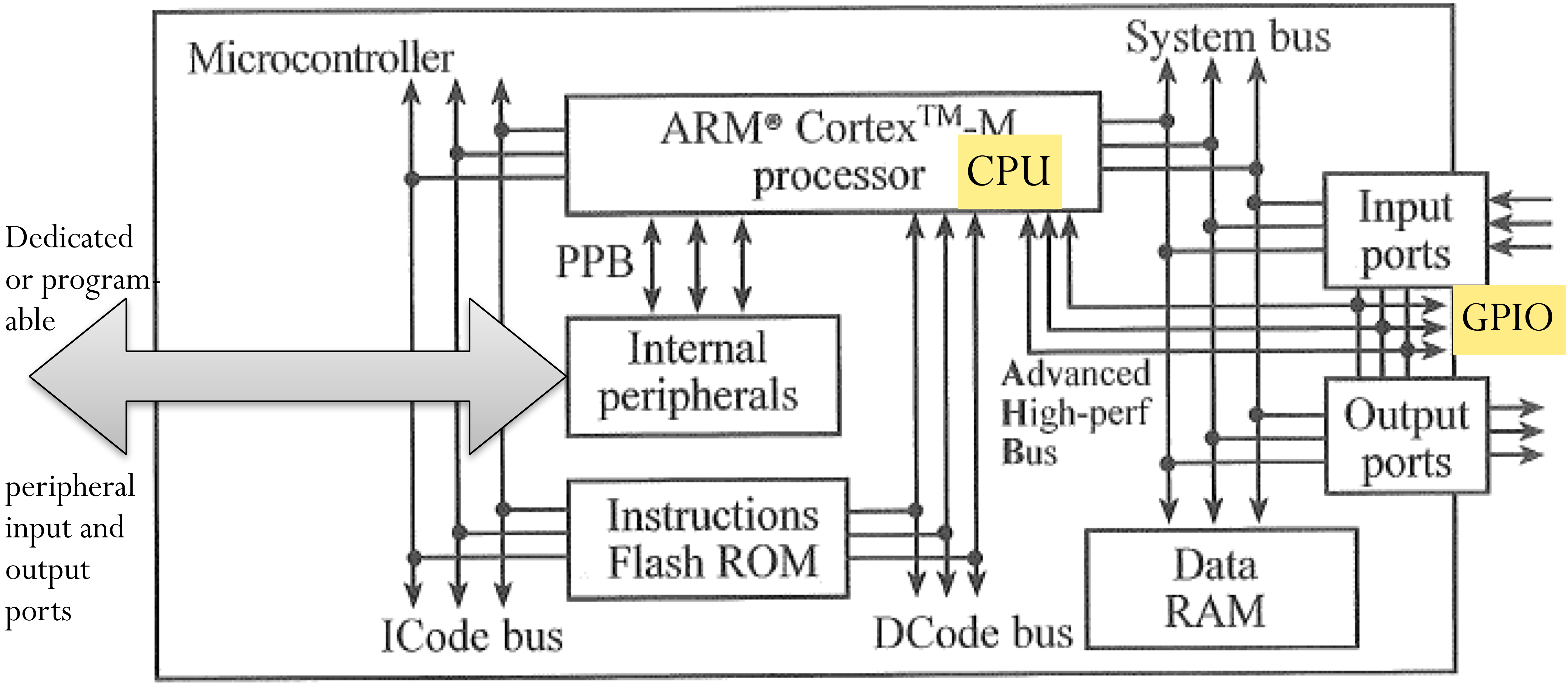
Roadmap

		ENCE361 Lecture, Tutorial & Lab Schedule – 2020					v. 20.3	Updated 09/05/2020
		Lec 1		Lec 2		Lec 3		Lab
Wk	Starting	Mon 1p		Wed 2p		Fri 2p		Mon 11a, Tue 9a, Tue 11a, Wed 11a
10	11 May	23	Profiling	26	Load Analysis	27	MCU Interfacing	
11	18 May	28	Memory structures	29	MCU memory types	30	Arm Arithm./ Logic ccts	**Project demos in usual lab slot**
12	25 May	31	The ARM ISA	32	ARM Assembly language	33	Revision & Exam Prep	**Project report & code due Fri 29 May**

Outline

- MCU Overview
  - CPU Datapath
    - ALU
    - Register File
    - Barrel Shifter

Overview of the Cortex-M4 MCU



Dedicated or programmable

peripheral input and output ports

From: Valvano, "Introduction to the ARM Cortex-M Microcontrollers", 2017 [4]. Annotations by Steve Weddell.

Overview of a CPU

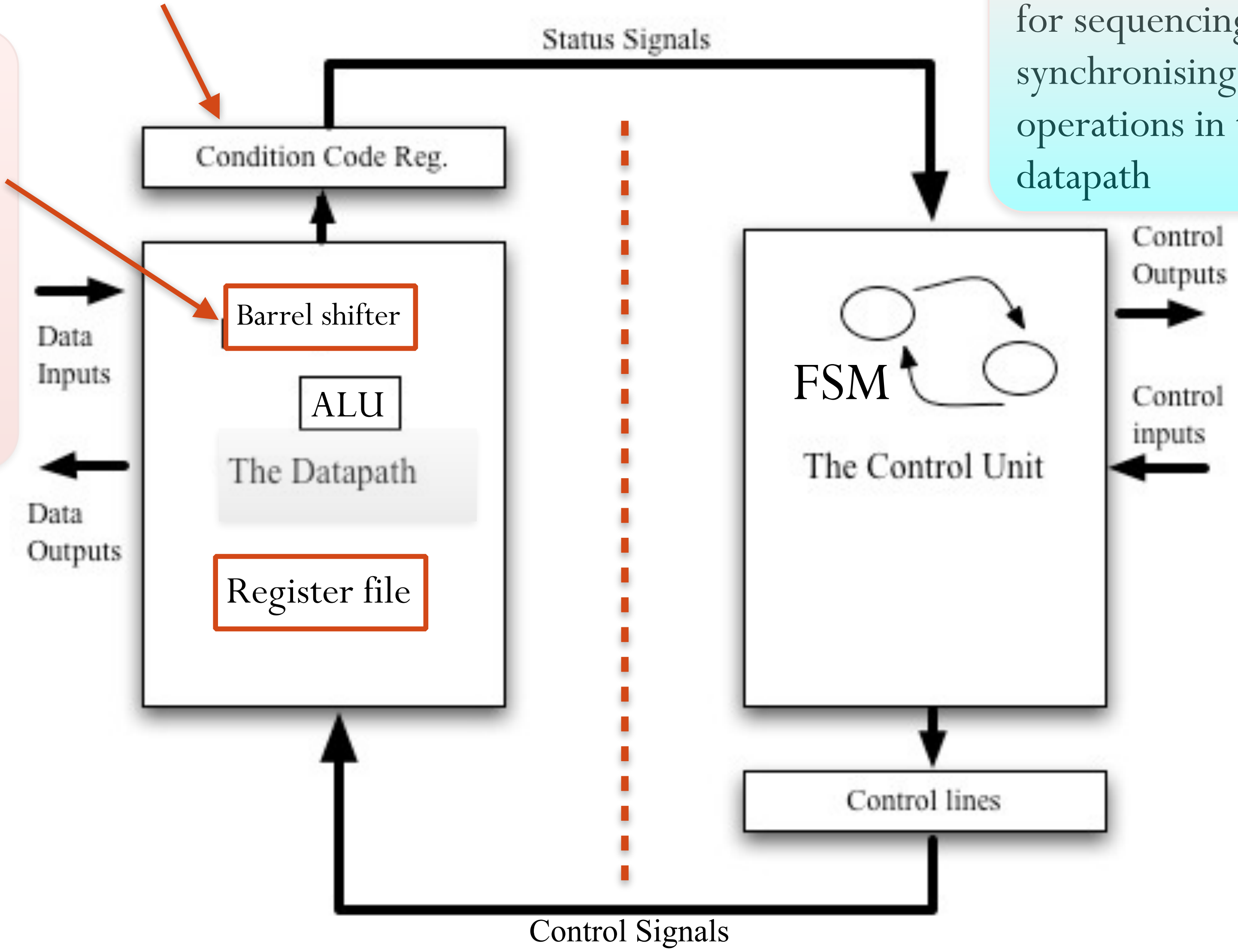
The **Datapath** comprises:

- Arithmetic & Logic Unit (ALU).
- Register File.
- Barrel shifter.
- Associated interface circuitry.

Supported condition code registers, such as N, V, C, and Z

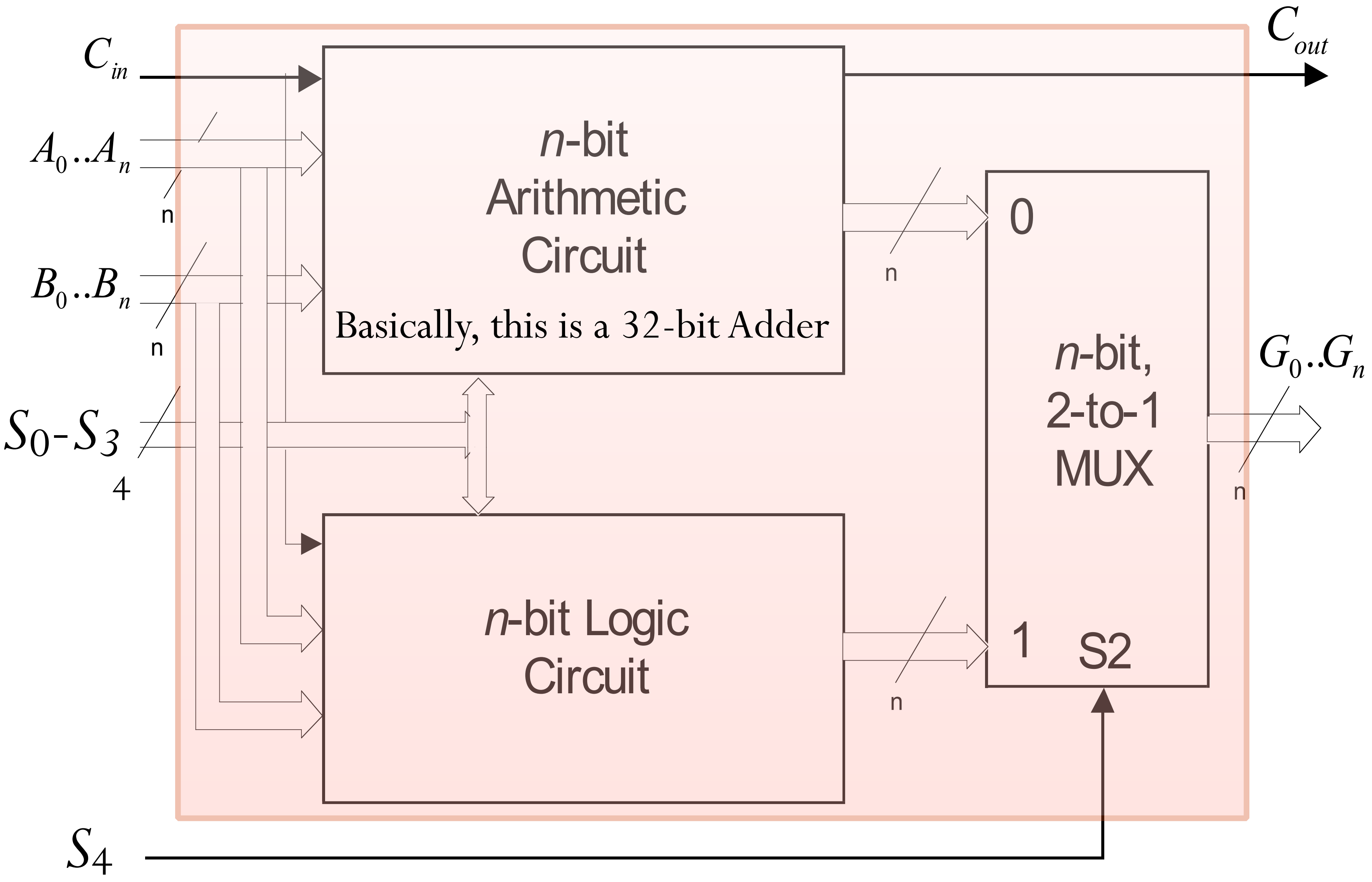
The **Control Unit** comprises:

Control circuits for sequencing/synchronising operations in the datapath



Ciaran Moore
ciaran.moore@canterbury.ac.nz

Arithmetic and Logic Unit (ALU)



Basically, this is a 32-bit Adder

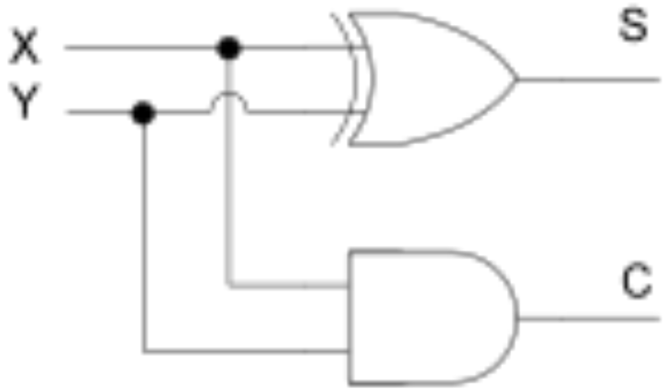
Ciaran Moore
ciaran.moore@canterbury.ac.nz

Arithmetic and Logic Unit (ALU)

- There are three components in an ALU:
  - Adder – performs arithmetic.
  - Logic circuit – A “forest” of logic gates that can perform binary (AND, OR, XOR) and unitary (NOT) operations on the inputs.
  - A 2-to-1 Multiplexer (Mux), that connects the output to either the adder or the logic circuit.
- The behaviour of these components is determined by Control Lines ($S_{0\text{-}4}$)

Adders

X	Y	C	S
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0



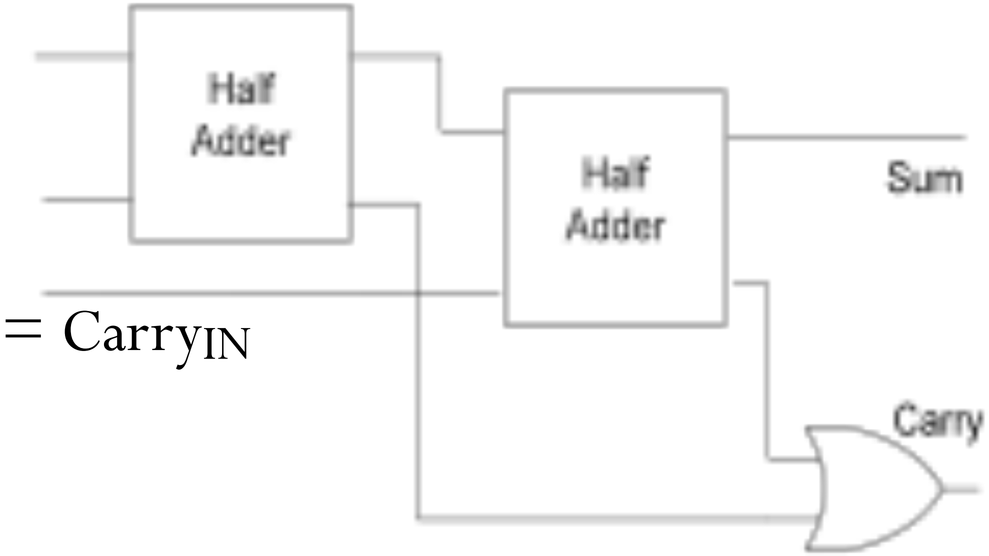
Half-adder (no provision for previous carry)

X	Y	Z	Carry	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

X

Y

Z = Carry$_{IN}$



Subtraction is performing by inverting one of the operands.

ALU Control Lines

S_5	S_4	S_3	S_2	S_1	S_0	**ALU output** *(G)*	
0	0	0	1	0	0	A and B	
0	0	1	0	0	0	A and not B	
0	0	1	0	0	1	A xor B	
0	1	1	0	0	1	A plus not B plus carry	* A - B
0	1	0	1	1	0	A plus B plus carry	A + B + carry
1	1	0	1	1	0	not A plus B plus carry	* B - A with borrow
0	0	0	0	0	0	A	
0	0	0	0	0	1	A or B	
0	0	0	1	0	1	B	
0	0	1	0	1	0	not B	
0	0	1	1	0	0	zero	

ARM-2 ALU Functions [Furber, 2000]

ALU Control Lines

- Two's complement is used to implement subtractions.
- A set "carry" bit is used to convert one's complement to two's complement.
- Inputs A and B can be inverted with XOR gates (A XOR 1 = !A).

Ciaran Moore
ciaran.moore@canterbury.ac.nz

ARM-6 ALU

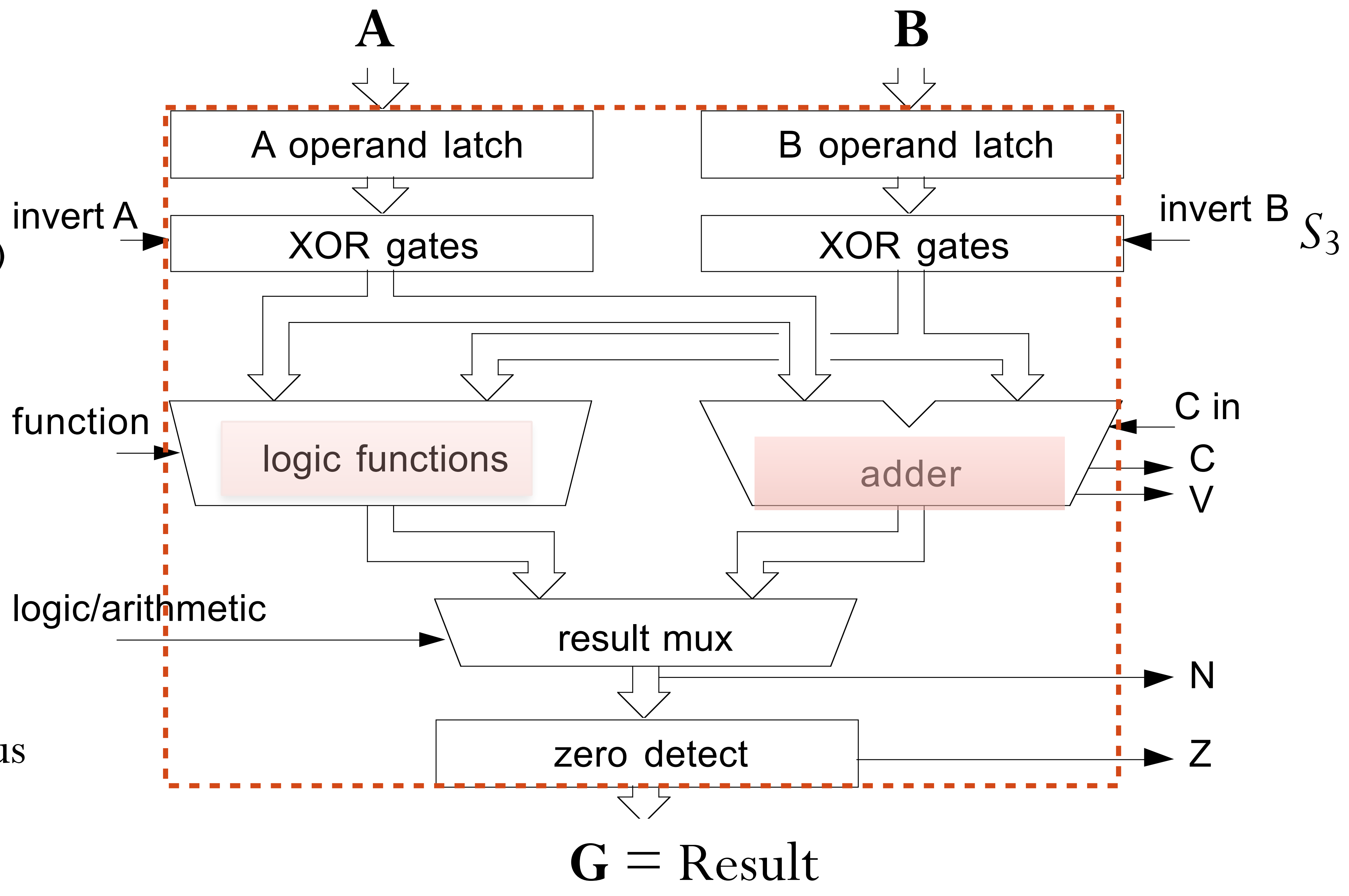
(not used in ARM2 ALU)

Select Lines $S_0..S_2$

S_3

S_4

(shown in the previous slide)



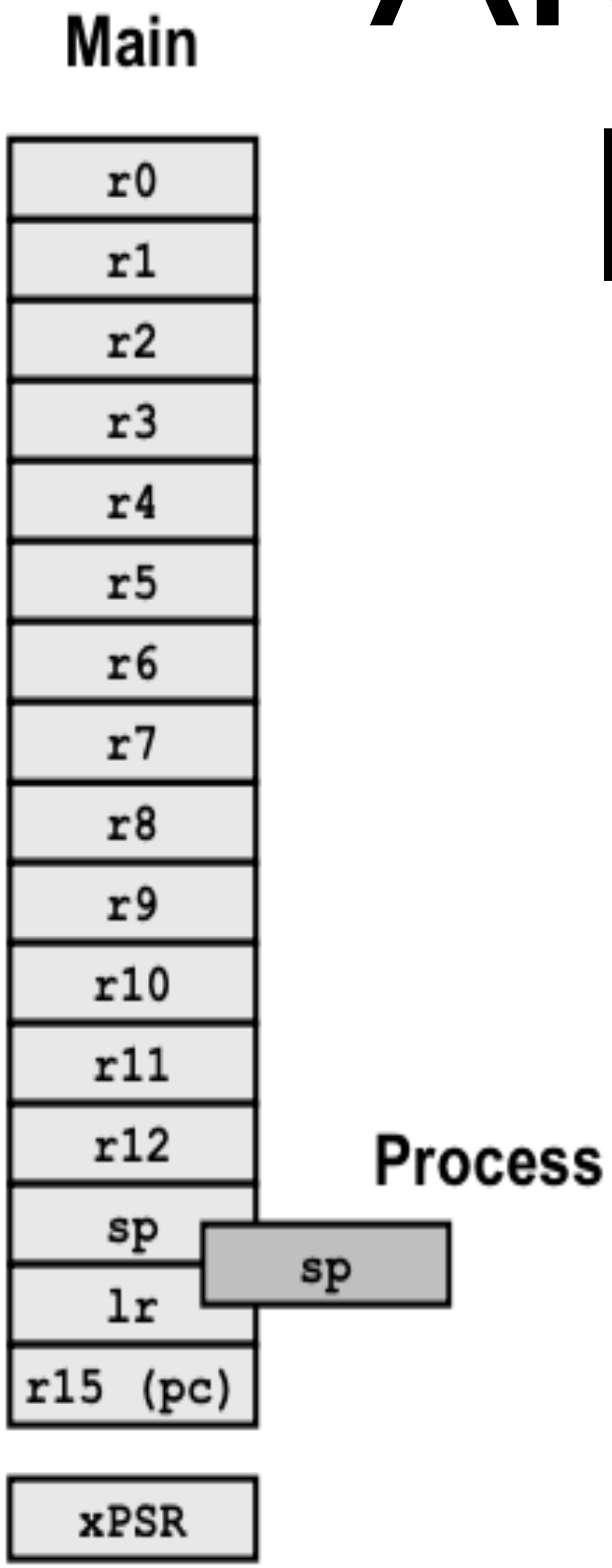
$\mathbf{G}$ = Result

Ciaran Moore
ciaran.moore@canterbury.ac.nz

ALU Flags

- ALU flags are stored in the Condition Code Register (CCR).
- They describe the result, G, returned by the ALU.
- C: Carry
- N: Negative
- V: Overflow
- Z: Zero
- Q: Saturation (similar to Overflow)

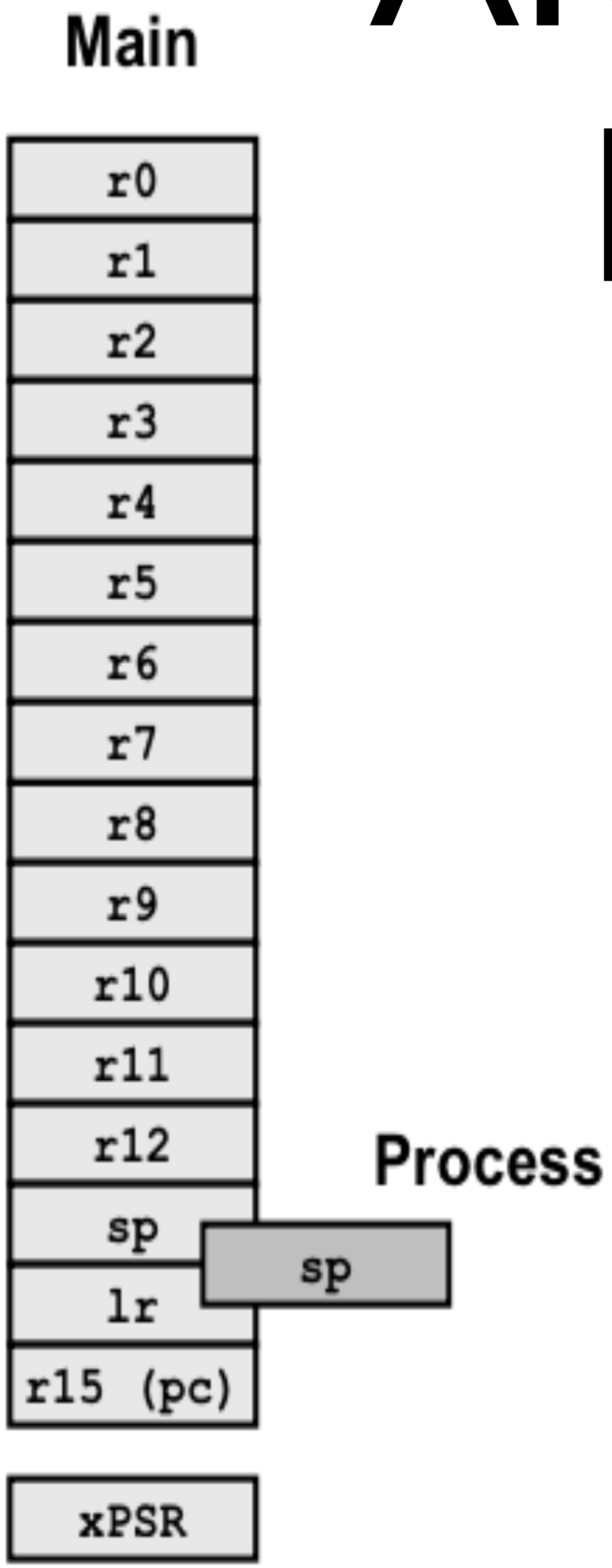
ARM Cortex-M4 Register File



- The register file provides source and destination operands for the ALU.
- All registers are 32 bits.
- Typically implemented as SRAM with separate read & write ports to maximise speed.

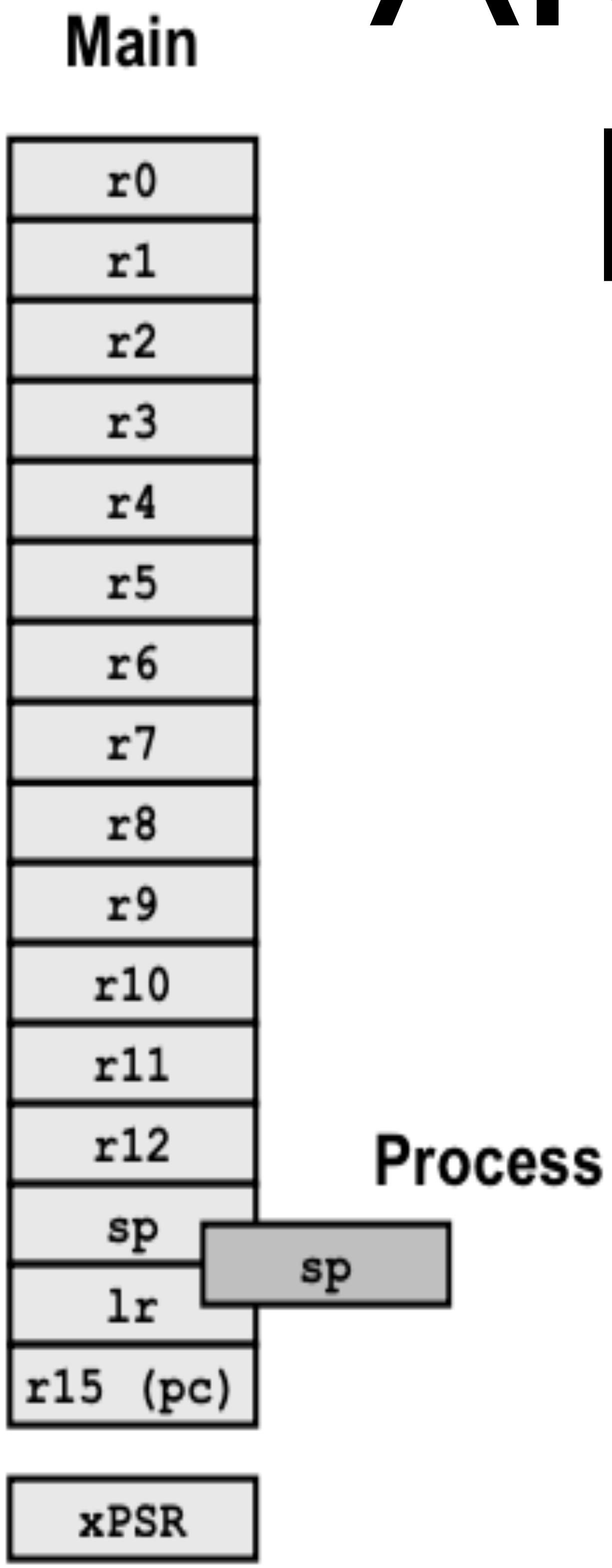
Ciaran Moore
ciaran.moore@canterbury.ac.nz

ARM Cortex-M4 Register File



- r0-r12 are general purpose registers.
- Used for storing intermediate variables, pointers and function arrays.
- r0 always stores the value returned by a function.

ARM Cortex-M4 Register File

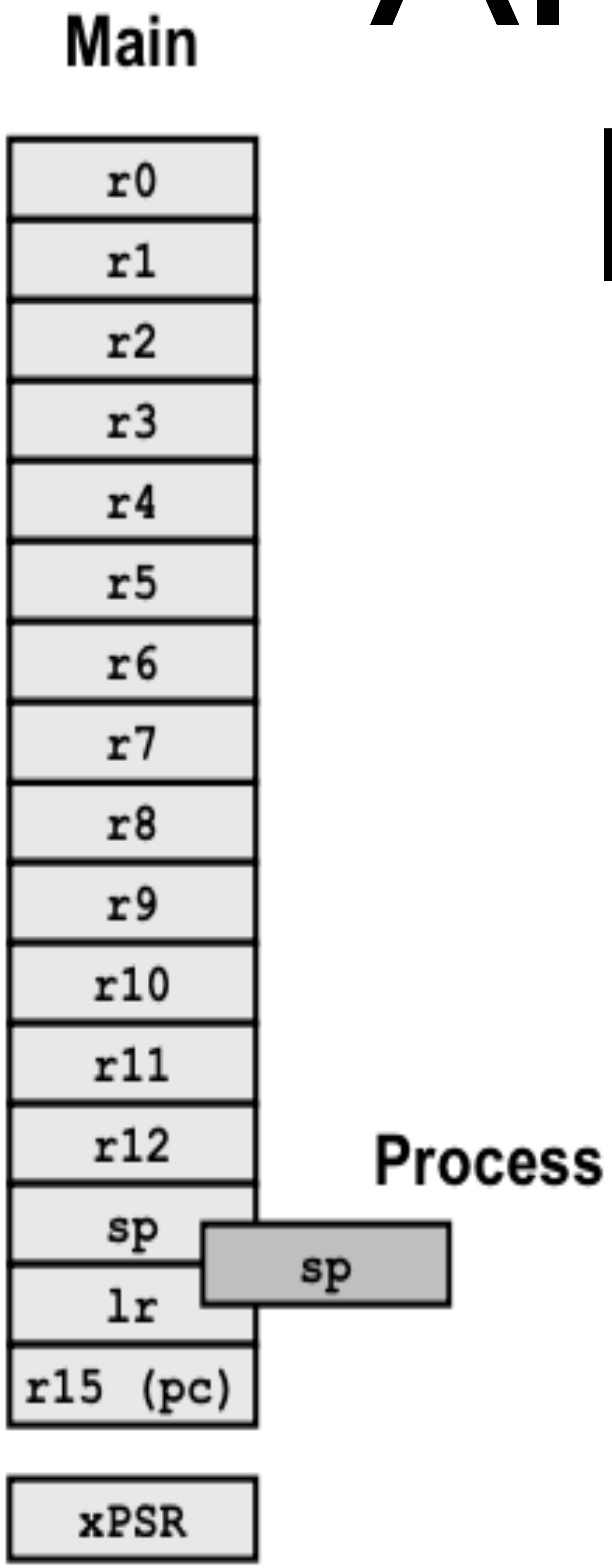


- sp = Stack Pointer. Points to top item on the stack. sp is **banked**. The *process* sp is used for exceptions.

- Some MCUs (e.g. ARM7 TDMI) use banked (i.e. duplicate) versions of the register file for different processing modes: so there's a version of the register file for "main" processing, but also one for interrupts, etc. This is done to reduce interrupt latency.

- ARM Cortex-M CPUs only bank sp because the NVIC allows fast servicing of interrupts through tight integration with the CPU.

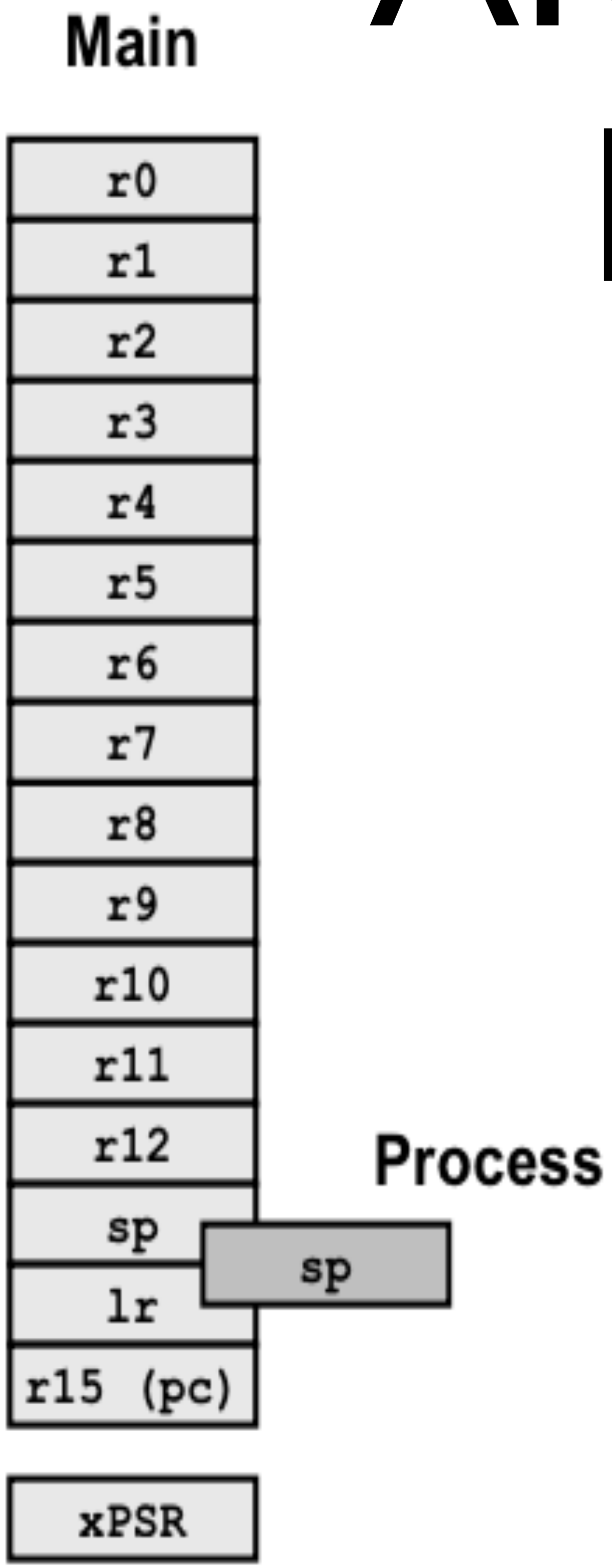
ARM Cortex-M4 Register File



- lr = Link Register. Stores return address for where execution should continue once a subroutine or function call is complete.
- pc = Program Counter. Stores the address of the instruction currently being executed.

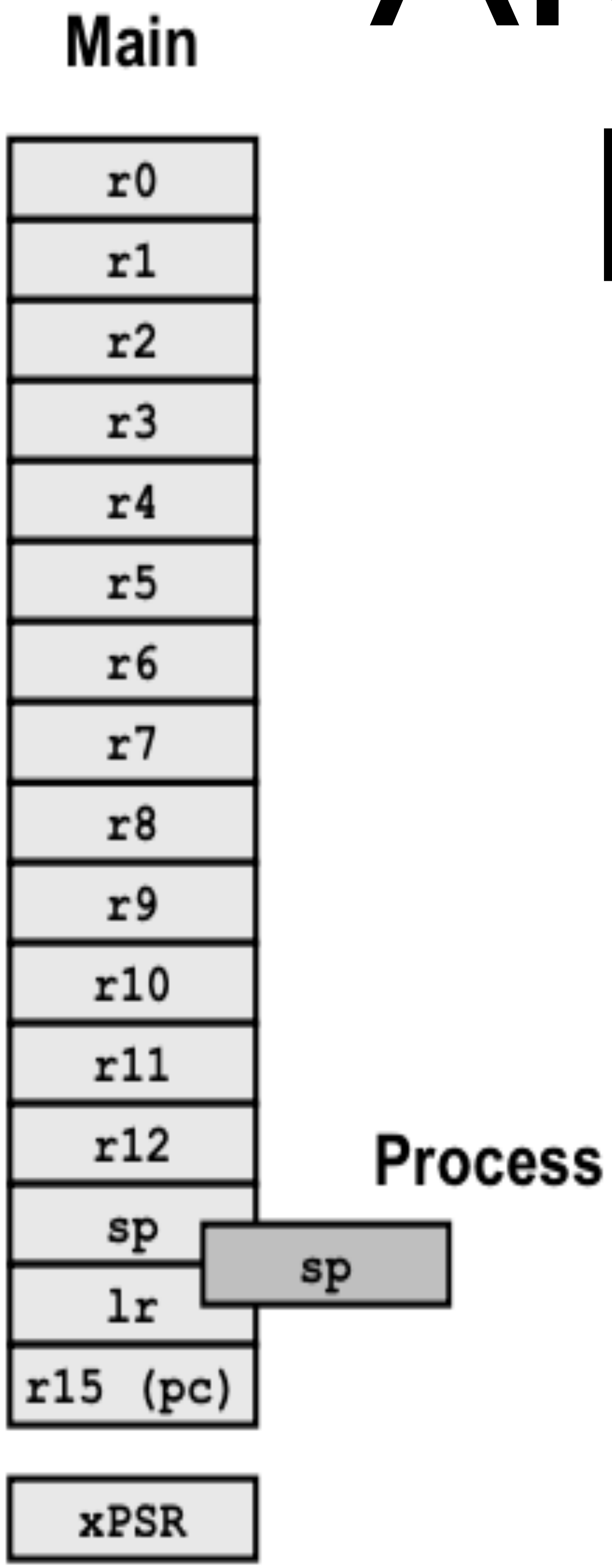
Ciaran Moore
ciaran.moore@canterbury.ac.nz

ARM Cortex-M4 Register File



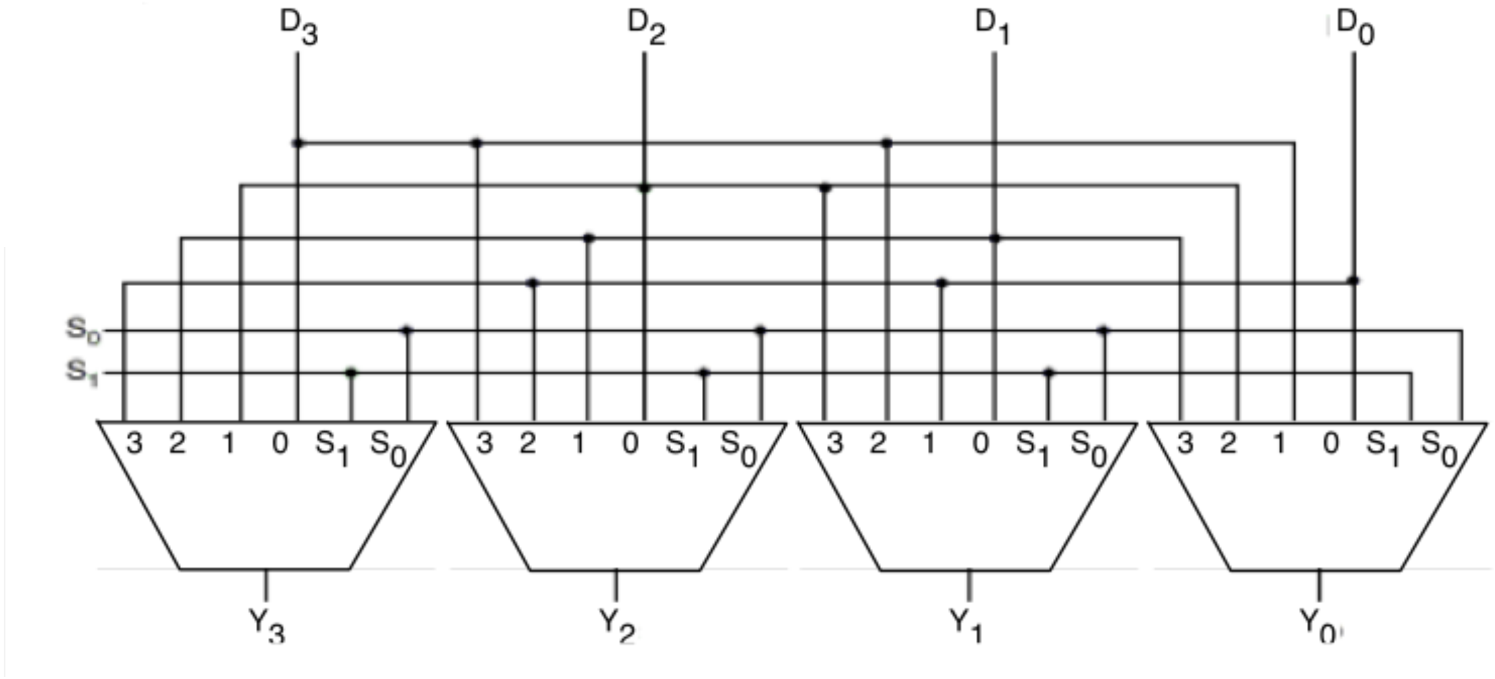
- xPSR = Program Status Register.
    - Application PSR: Stores N, Z, C, V, Q flags.
    - Interrupt PSR: Stores ISR number.
    - Execution PSR: Stores information about interrupt-continuable instructions.

ARM Cortex-M4 Register File



- r0-r3 and r12, along with sp, lr and pc, are pushed to the stack when an interrupt is serviced.

Barrel Shifter



Function Table for 4-Bit Barrel Shifter

Select		Output				
S_1	S_0	Y_3	Y_2	Y_1	Y_0	Operation
0	0	D_3	D_2	D_1	D_0	No rotation
0	1	D_2	D_1	D_0	D_3	Rotate one position
1	0	D_1	D_0	D_3	D_2	Rotate two positions
1	1	D_0	D_3	D_2	D_1	Rotate three positions

[Mano & Kime]

Barrel Shifter

- A *shift module* allows a group of data bits within a register to be shifted, either one place to the left or right.
- A *barrel shifter* allows left or right shifts by an arbitrary number of bits without iteration.
- A barrel shifter module is typically part of the data path within a CPU.
- Bidirectional shifts are possible due to the rotational characteristic of barrel shifters. For example, for a 4-bit barrel shifter shifting right by 1 is equivalent to shifting left by 3.

Summary

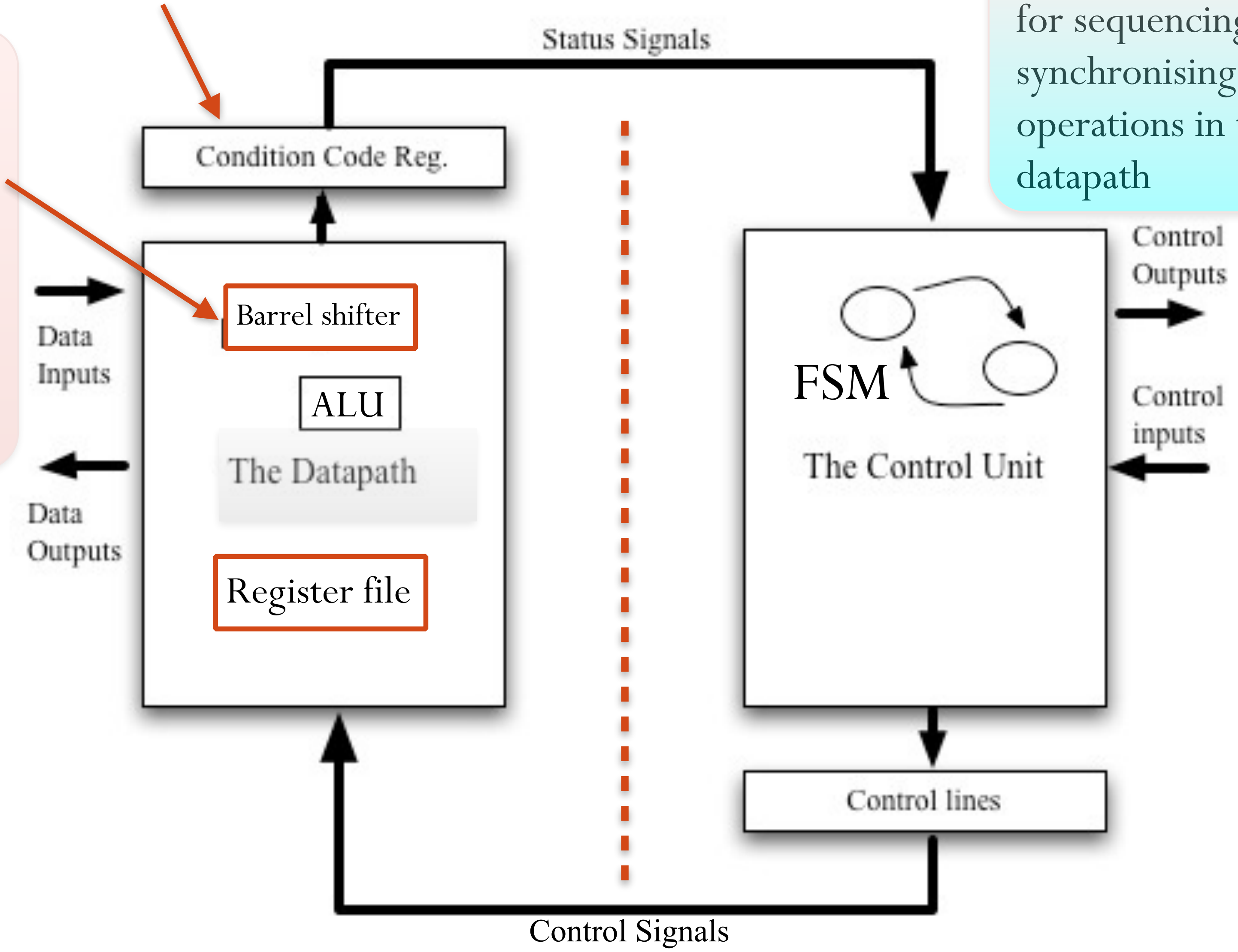
The **Datapath** comprises:

- Arithmetic & Logic Unit (ALU).
- Register File.
- Barrel shifter.
- Associated interface circuitry.

Supported condition code registers, such as N, V, C, and Z

The **Control Unit** comprises:

Control circuits for sequencing/synchronising operations in the datapath



Ciaran Moore
ciaran.moore@canterbury.ac.nz

Instruction Set Architectures

ENCE361: Design & Architecture: Lecture Block 3

Roadmap

		ENCE361 Lecture, Tutorial & Lab Schedule – 2020					v. 20.3	Updated 09/05/2020
		Lec 1		Lec 2		Lec 3		Lab
Wk	Starting	Mon 1p		Wed 2p		Fri 2p		Mon 11a, Tue 9a, Tue 11a, Wed 11a
10	11 May	23	Profiling	26	Load Analysis	27	MCU Interfacing	
11	18 May	28	Memory structures	29	MCU memory types	30	Arm Arithm./ Logic ccts	**Project demos in usual lab slot**
12	25 May	31	The ARM ISA	32	ARM Assembly language	33	Revision & Exam Prep	**Project report & code due Fri 29 May**

Outline

- Instruction Set Architecture
- Hardware Abstraction Layer
- Types of ISA
- Examples of Instructions
- ARM ISAs

~~Summary~~ Review

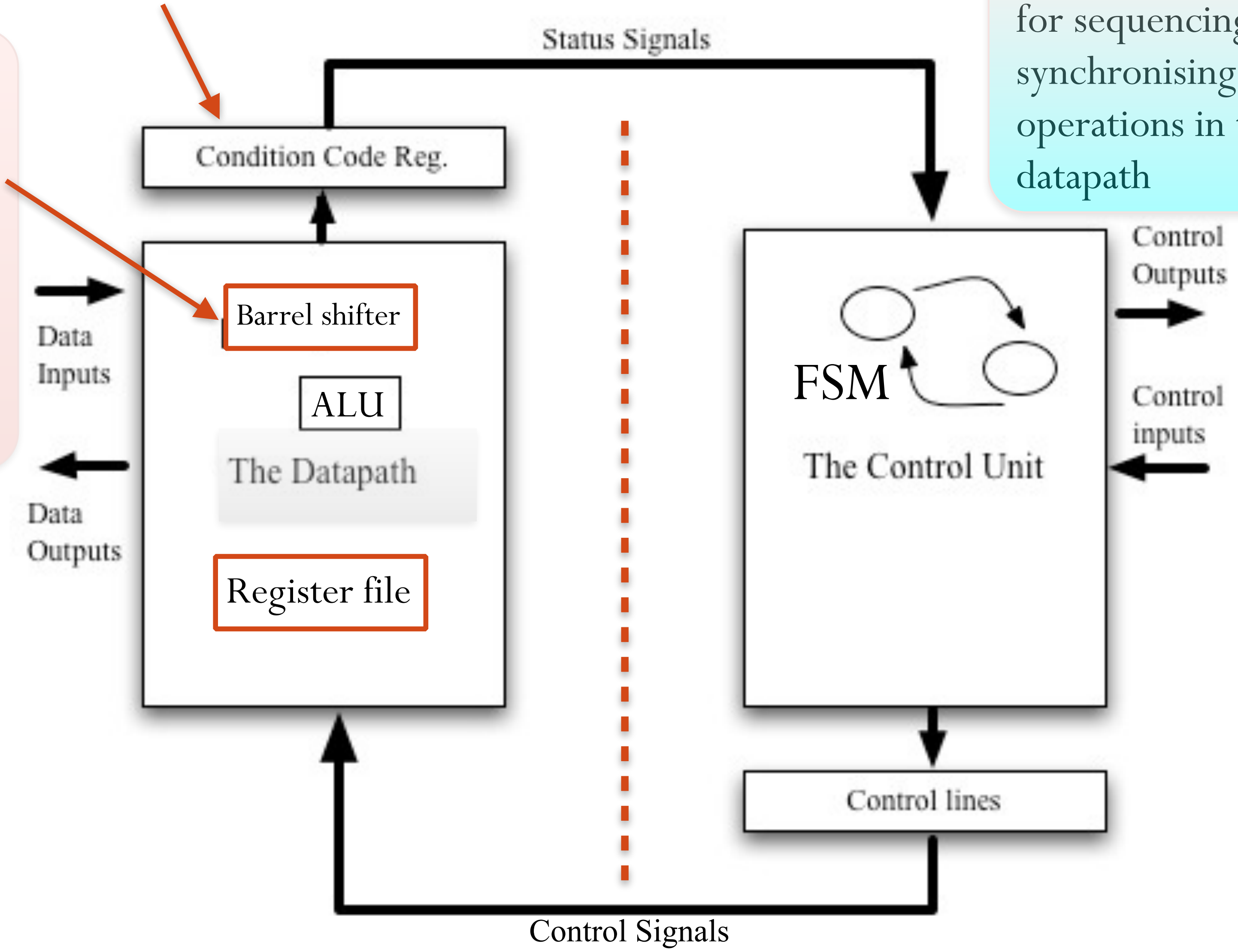
The **Datapath** comprises:

- Arithmetic & Logic Unit (ALU).
- Register File.
- Barrel shifter.
- Associated interface circuitry.

Supported condition code registers, such as *N*, *V*, *C*, and *Z*

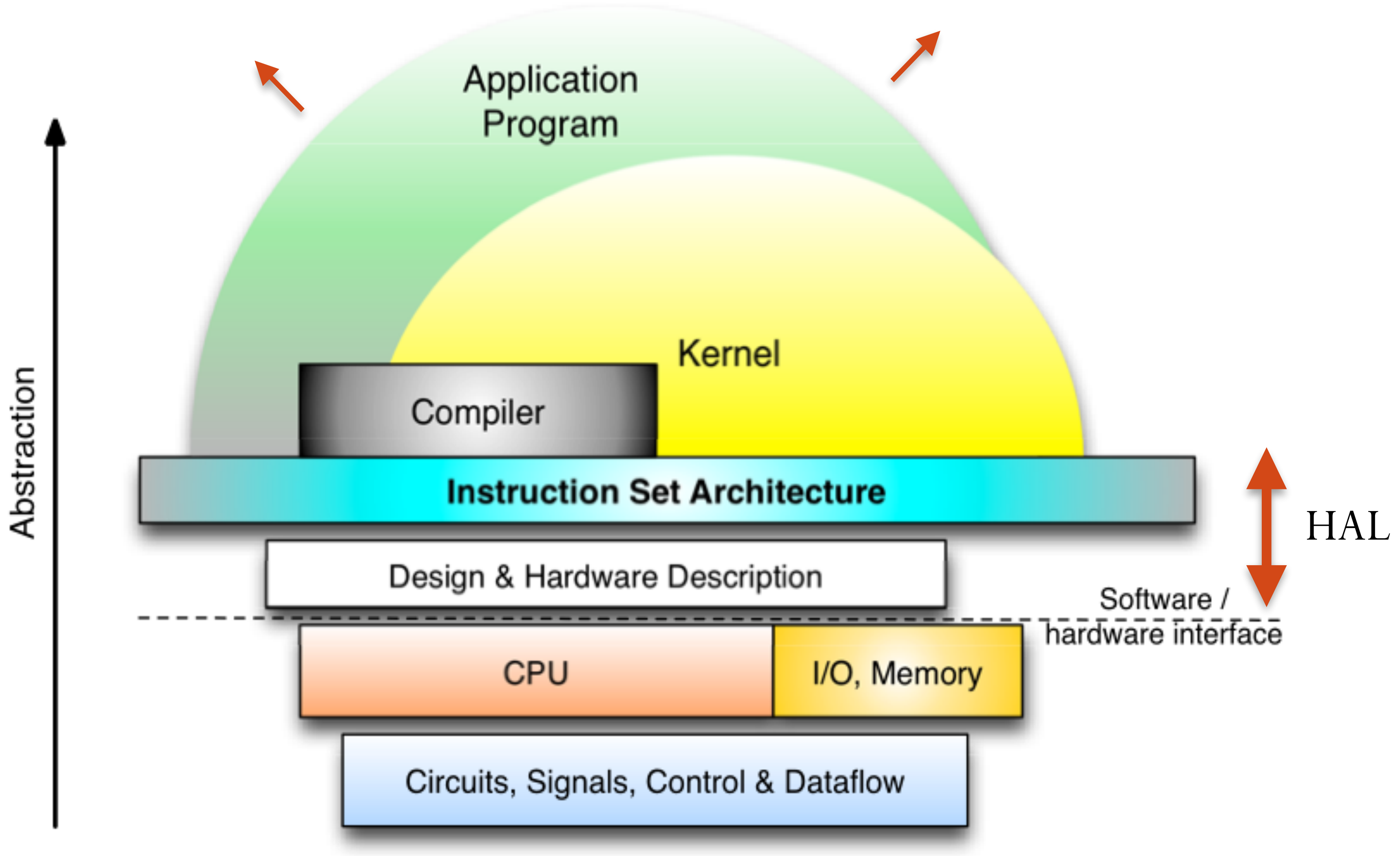
The **Control Unit** comprises:

Control circuits for sequencing/ synchronising operations in the datapath



Ciaran Moore
ciaran.moore@canterbury.ac.nz

Instruction Set Architecture

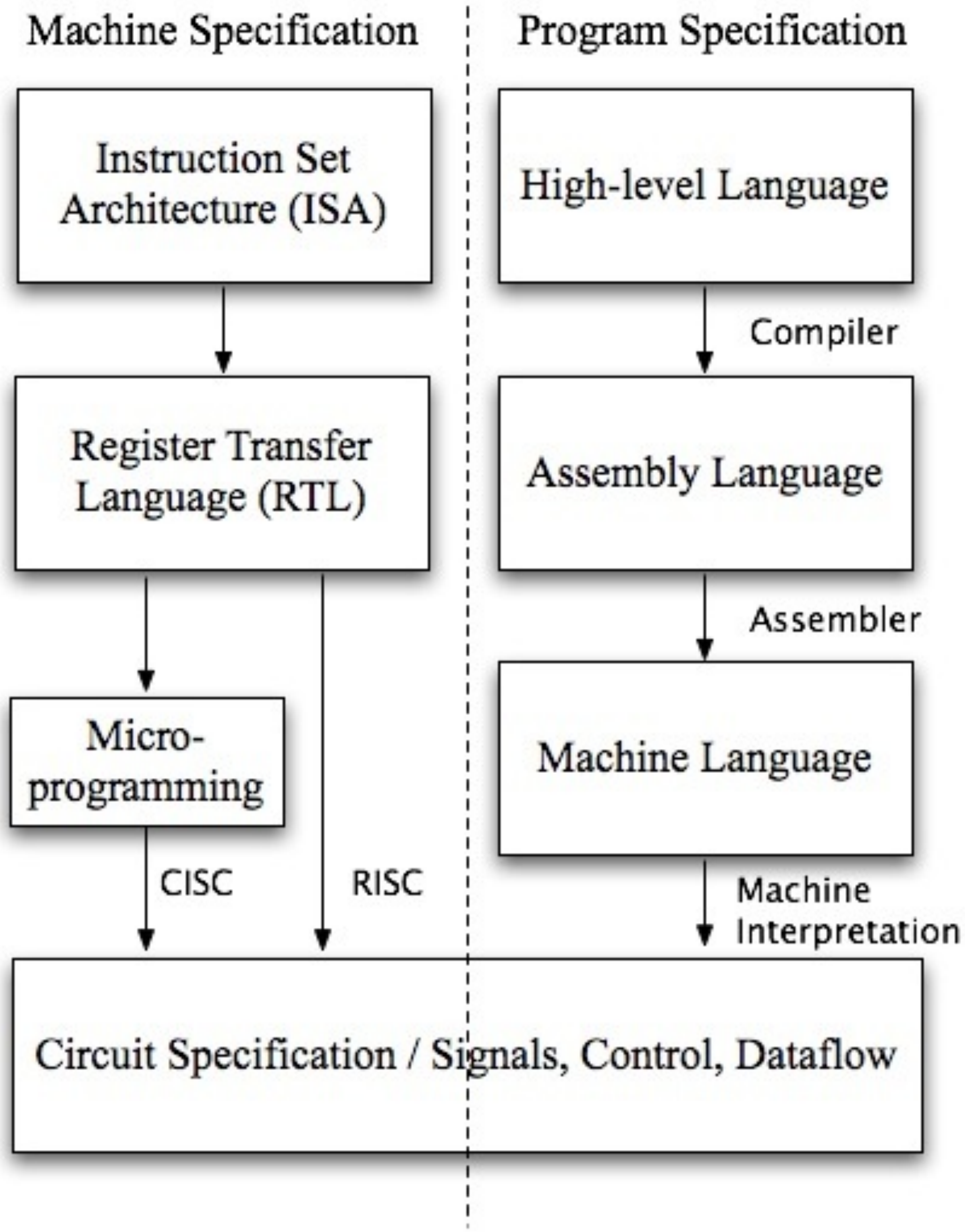

Application
Program
Kernel
Compiler
Instruction Set Architecture
Design & Hardware Description
CPU
I/O, Memory
Circuits, Signals, Control & Dataflow
Abstraction
HAL
Software /
hardware interface

HAL: hardware abstraction layer

Ciaran Moore
ciaran.moore@canterbury.ac.nz

Hardware Abstraction Layer

- *Program specification* represents a top-down program (software) abstraction.
- *Machine specification* represents a top-down machine (hardware/firmware) abstraction.
- Reduced Instruction Set Computers (RISC) use hardware state machines for control.
- Complex Instruction Set Computers (CISC) use software (micro-coded) state machines for control.



Types of ISA

- Most computations (a.k.a. *operations*) require three *operands*: two for the sources of data, and one for the destination.
- If A and B are source registers and D is a destination register, then in *register transfer language* (RTL):

$$D \leftarrow A \, \text{operation} \, B$$

Types of Instruction

- The number of operands per instruction varies and is related to the number of address and data buses in an architecture:
    - **3 address instructions:** Two sources and one destination operand.
    - Used in register-to-register (load-store) architectures.
    - e.g. `ADD R3, R1, R2` (RTL: R3 ← R1 + R2) [MIPS, ARM]

Types of Instruction

- The number of operands per instruction varies and is related to the number of address and data buses in an architecture:
  - **2 address instructions:** One source and one destination operand are explicitly stated. Second source is either the destination or implied by the instruction.
  - e.g. `ABA` **;** (RTL: A ← A + B)
    [CPU-12 by Freescale/NXP]

Types of Instruction

- The number of operands per instruction varies and is related to the number of address and data buses in an architecture:
  - **1 address instructions:** Use an *implied operand* to double as a source AND destination address. A common architecture for *single accumulator* CPUs.
  - e.g. `ADD #$F0` (RTL: Acc ← Acc + $F0) [M68HC05 by Freescale/NXP]

Types of Instruction

- The number of operands per instruction varies and is related to the number of address and data buses in an architecture:
  - **0 address instructions:** Operands are stored in pre-defined order on the stack.
  - e.g. `PSHA, PSHB, PULA` etc.

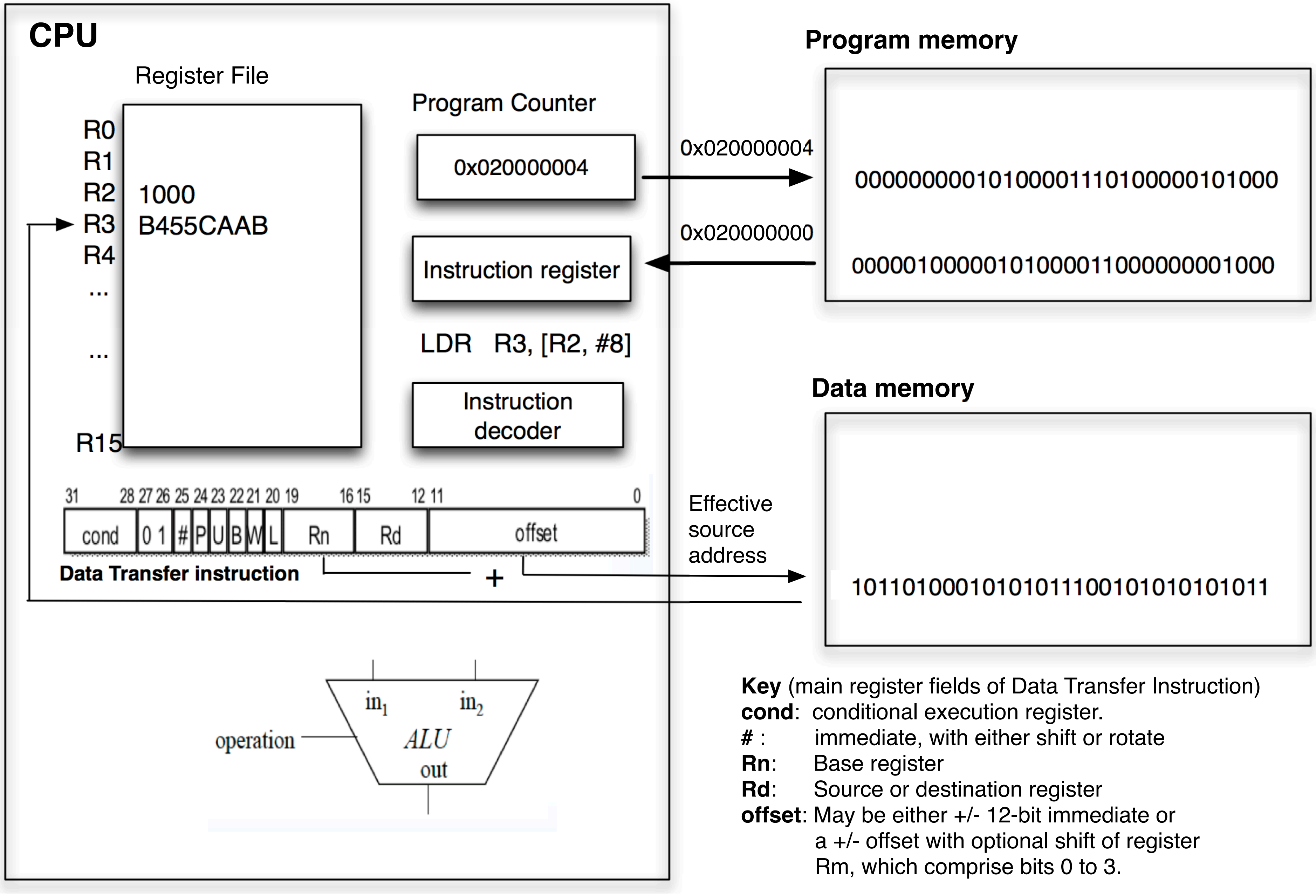
Glossary of terms used to describe ISAs

- ***Operand***: Additional information that an instruction may require to complete its operation. Distinct from an instruction's ***operation*** (op) code. Operands can be a literal value, an address, the contents of a register, or a specific register.
- ***Effective address*** is a memory address given by the value in a register + some offset, given by an operand.
- ***Condition Code Register*** (CPSR or xPSR in ARM processors) contains a series of single-bit flags used to provide information about the datapath to the control unit. Branch instructions (`BNE`, `CMP`) ***test*** bits in the CCR and output control signals accordingly.

Glossary of terms used to describe ISAs

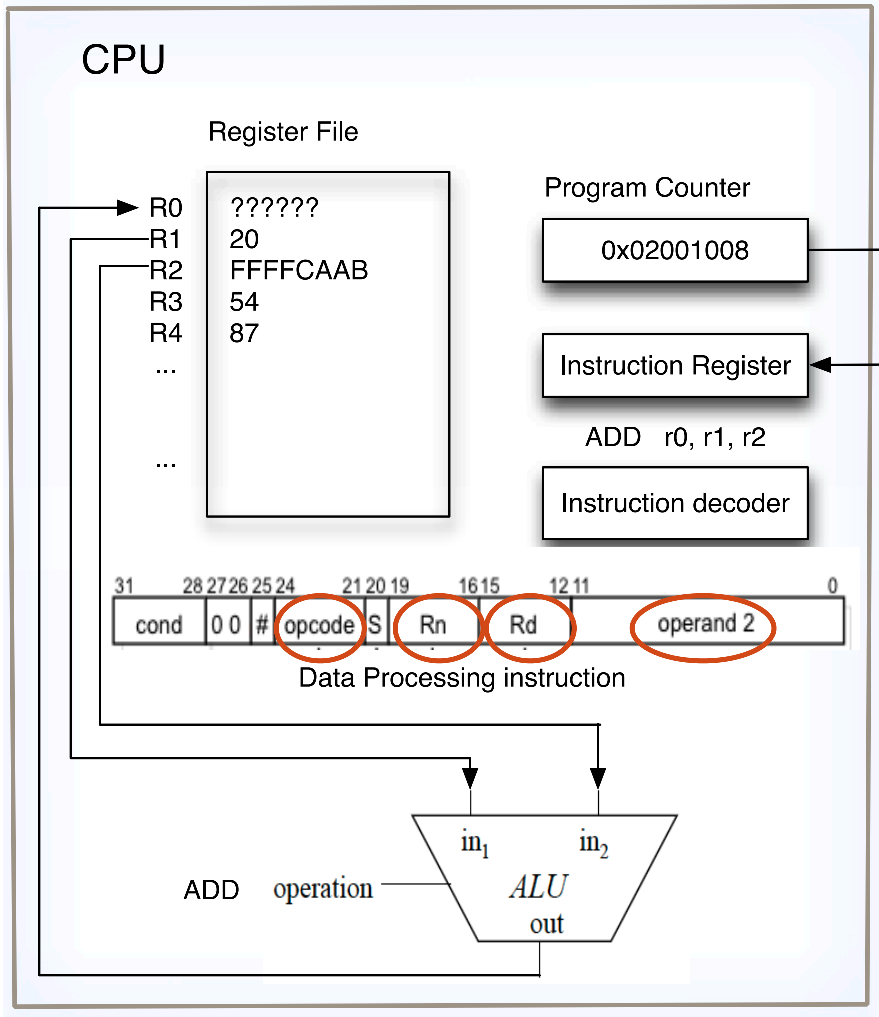
- The CPU ***datapath*** comprises circuit modules that provide access to the data (operands) to be executed by each instruction. Comprises the ***ALU, register file, barrel shifter*** and associated circuitry.
- The CPU ***control unit*** is effectively a state machine, designed to execute each instruction stage in a specific sequence, e.g.:
  - ***Fetch, Decode, Execute***
    (for instructions with implicit operands (CISC))
  - ***Fetch, Decode, Read Op, Execute, Store Result***
    (for instructions with explicit operands (RISC))

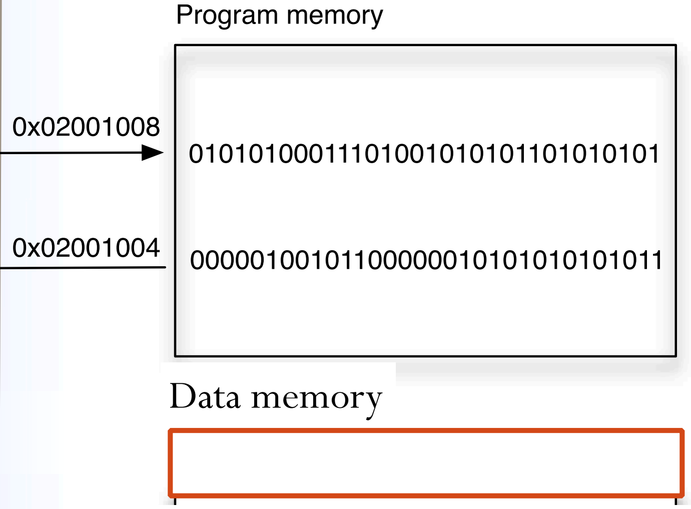
Load Data Instructions

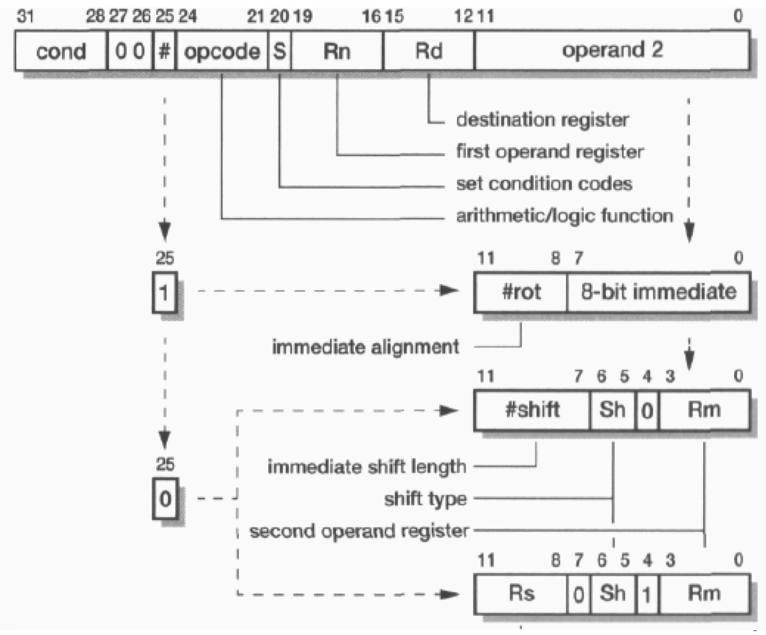


Key (main register fields of Data Transfer Instruction)
cond: conditional execution register.
: immediate, with either shift or rotate
Rn: Base register
Rd: Source or destination register
offset: May be either +/- 12-bit immediate or a +/- offset with optional shift of register Rm, which comprise bits 0 to 3.

Data Processing Instructions

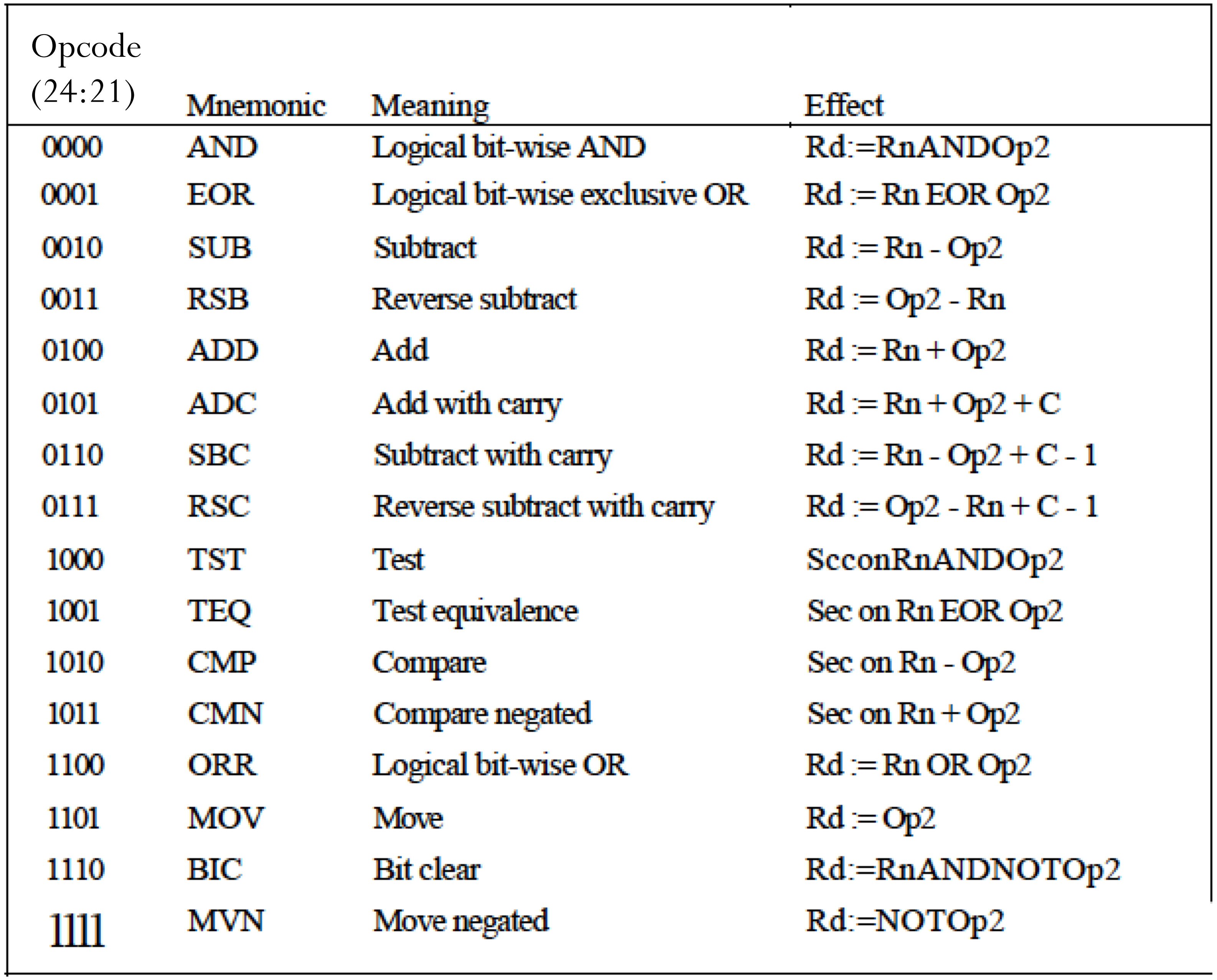






Data processing instructions

Data Processing Instructions



Opcode (24:21)	Mnemonic	Meaning	Effect
0000	AND	Logical bit-wise AND	Rd:=RnANDOp2
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	ScconRnANDOp2
1001	TEQ	Test equivalence	Sec on Rn EOR Op2
1010	CMP	Compare	Sec on Rn - Op2
1011	CMN	Compare negated	Sec on Rn + Op2
1100	ORR	Logical bit-wise OR	Rd := Rn OR Op2
1101	MOV	Move	Rd := Op2
1110	BIC	Bit clear	Rd:=RnANDNOTOp2
1111	MVN	Move negated	Rd:=NOTOp2

ARM – Core Features

- Load-Store Architecture: separate instructions are used to load and store operands in registers.
- All instructions are fixed length (32-bit).
- Three-address instruction format (2 source, 1 destination).
- All instructions provide conditional execution.
- Most instructions are performed in a single clock cycle.

ARM Cortex-M Series ISAs

- ARM: The original ARM ISA was comprised exclusively of 32-bit instructions. Together these made a powerful ISA, but the rise of mobile (smart) phones placed pressure on code size.
- Almost all ARM instructions have three operands (two source, one destination).

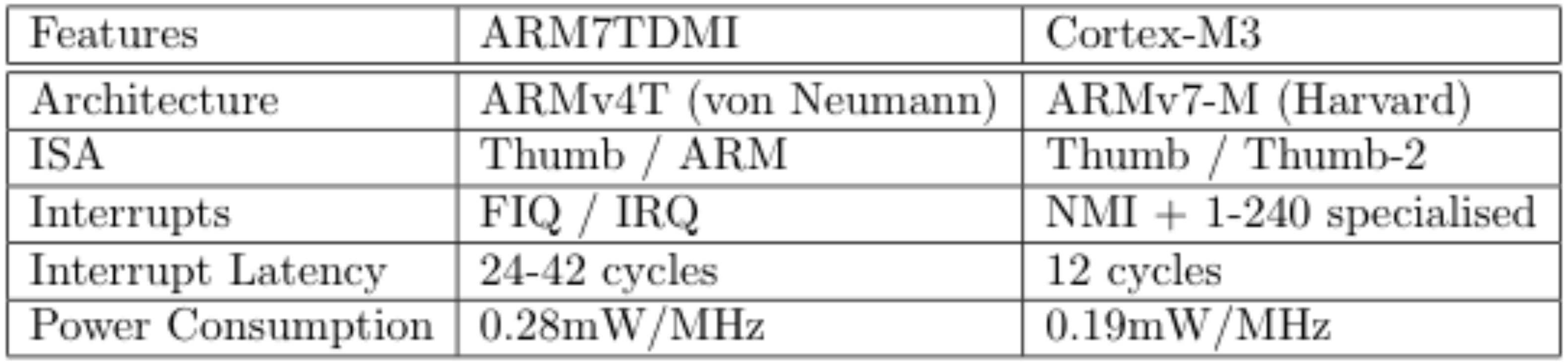
ARM Cortex-M Series ISAs

- Thumb: A second architecture with 16-bit compressed instructions. Hardware de-compressors convert 16-bit Thumb instructions to 32-bit ARM instructions.
- Although Thumb is a subset of ARM, it has higher code density, which makes it attractive where storage is limited.
- Thumb sacrifices conditional execution and explicit operands (most instructions have two operands).

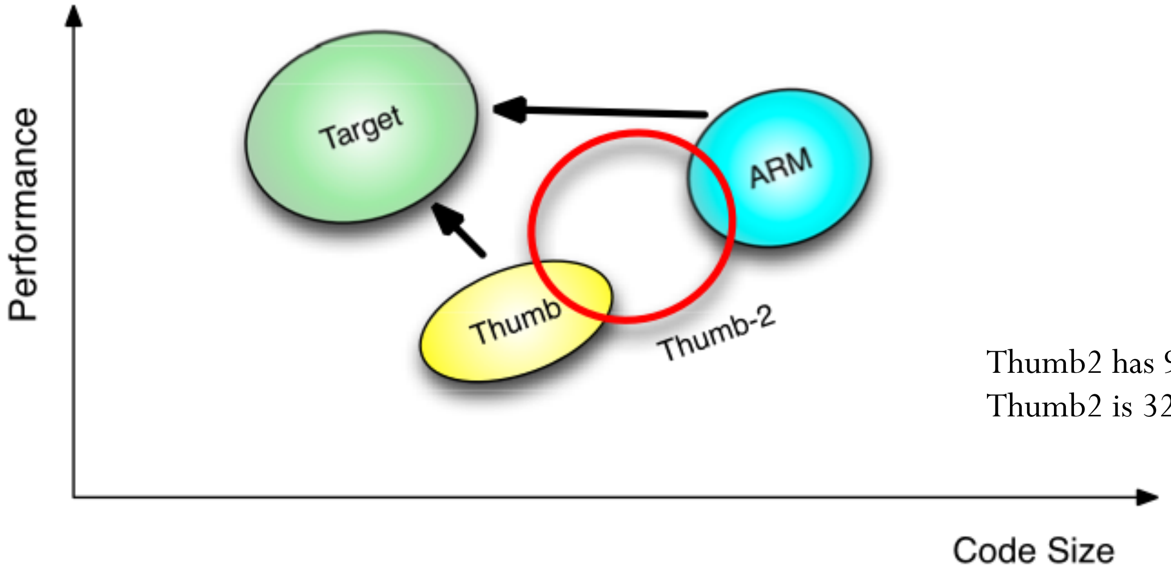
ARM Cortex-M Series ISAs

- Thumb2: An iteration of Thumb that supports almost as many instructions as the original ARM ISA.
- Like Thumb, most instructions are compressed to 16 bits (although some are 32-bit).
- Cortex-M series only supports Thumb and Thumb2 ISAs.

ARM Cortex-M Series ISAs



Features	ARM7TDMI	Cortex-M3
Architecture	ARMv4T (von Neumann)	ARMv7-M (Harvard)
ISA	Thumb / ARM	Thumb / Thumb-2
Interrupts	FIQ / IRQ	NMI + 1-240 specialised
Interrupt Latency	24-42 cycles	12 cycles
Power Consumption	0.28mW/MHz	0.19mW/MHz

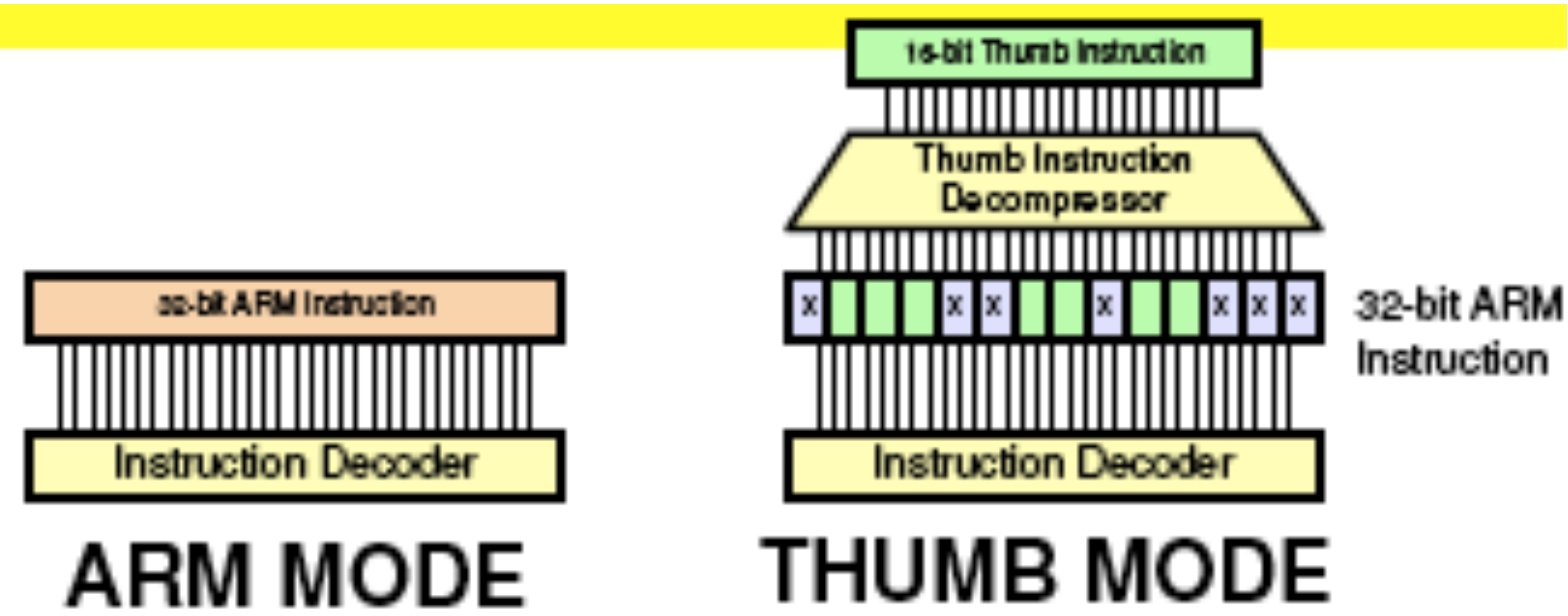


Thumb2 has 98% ARM performance
Thumb2 is 32% smaller than ARM

In terms of performance, the Cortex M3 vs M4 are very similar: https://en.wikipedia.org/wiki/ARM_Cortex-M

ARM Cortex-M Series ISAs

- ARM7TDMI integrates a real-time instruction de-compressor when executing instructions. Cortex-M3/4 only support Thumb and Thumb2.
- Thumb(2) instructions are de-compressed and expanded into 32-bit instructions. **The CPU always executes 32-bit instructions.**
- An application can switch between ARM and Thumb modes at any time using the .ARM. .THUMB or .THUMB2 pseudo-ops.



Summary

- Instruction Set Architecture
- Hardware Abstraction Layer
- Types of ISA
- Examples of Instructions
- ARM ISAs

ARM Assembly Language

ENCE361: Design & Architecture: Lecture Block 3

Roadmap

		ENCE361 Lecture, Tutorial & Lab Schedule – 2020					v. 20.3	Updated 09/05/2020
		Lec 1		Lec 2		Lec 3		Lab
Wk	Starting	Mon 1p		Wed 2p		Fri 2p		Mon 11a, Tue 9a, Tue 11a, Wed 11a
10	11 May	23	Profiling	26	Load Analysis	27	MCU Interfacing	
11	18 May	28	Memory structures	29	MCU memory types	30	Arm Arithm./ Logic ccts	**Project demos in usual lab slot**
12	25 May	31	The ARM ISA	32	ARM Assembly language	33	Revision & Exam Prep	**Project report & code due Fri 29 May**

Outline

- Why Use Assembler?
- Types of ARM Assembly Instructions:
  - Data Processing Instructions
  - Data Transfer/Movement Instructions
  - Control Flow Instructions
- In-Line Assembly Programming

Why use Assembler?

Assembler allows you to optimise code-critical sections of a program & test compiler efficiency.

But high-level languages like C have shorter dev time, are less cryptic and encourage programmers to modularise.

Why not use both? Start with C and add ASM subroutines or modules to boost performance as required.

ARM Assembly Language Programming

```
LABEL    OPCODE        OPERANDS      COMMENTS
         (mnemonic)
Start:
            MOV        r0, #15       ; immediate operand
            MOV        r1, #0x43     ;
            BL         Myfunc        ; call to subroutine
                                     ; and save retrn addr
Here:       B          Here          ; endless loop

Myfunc:
            ADD        r0, r0, r1    ; add two operands
            MOV        pc, lr        ; return from subr.
```

Destination operand

Types of ARM Assembly Instructions

- Data Processing Instructions
- Data Transfer/Movement Instructions
- Control Flow Instructions

Data Processing Instructions

- Use 32-bit operands (i.e. from registers).
- Give 32-bit register result.
- Use a 3-address architecture (2 source & 1 destination operand),
  e.g. `r0 = r1 + r2`
- Support both signed (2's comp) and unsigned data.
- Status bits are **optionally** updated by appending 'S' to mnemonic.

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

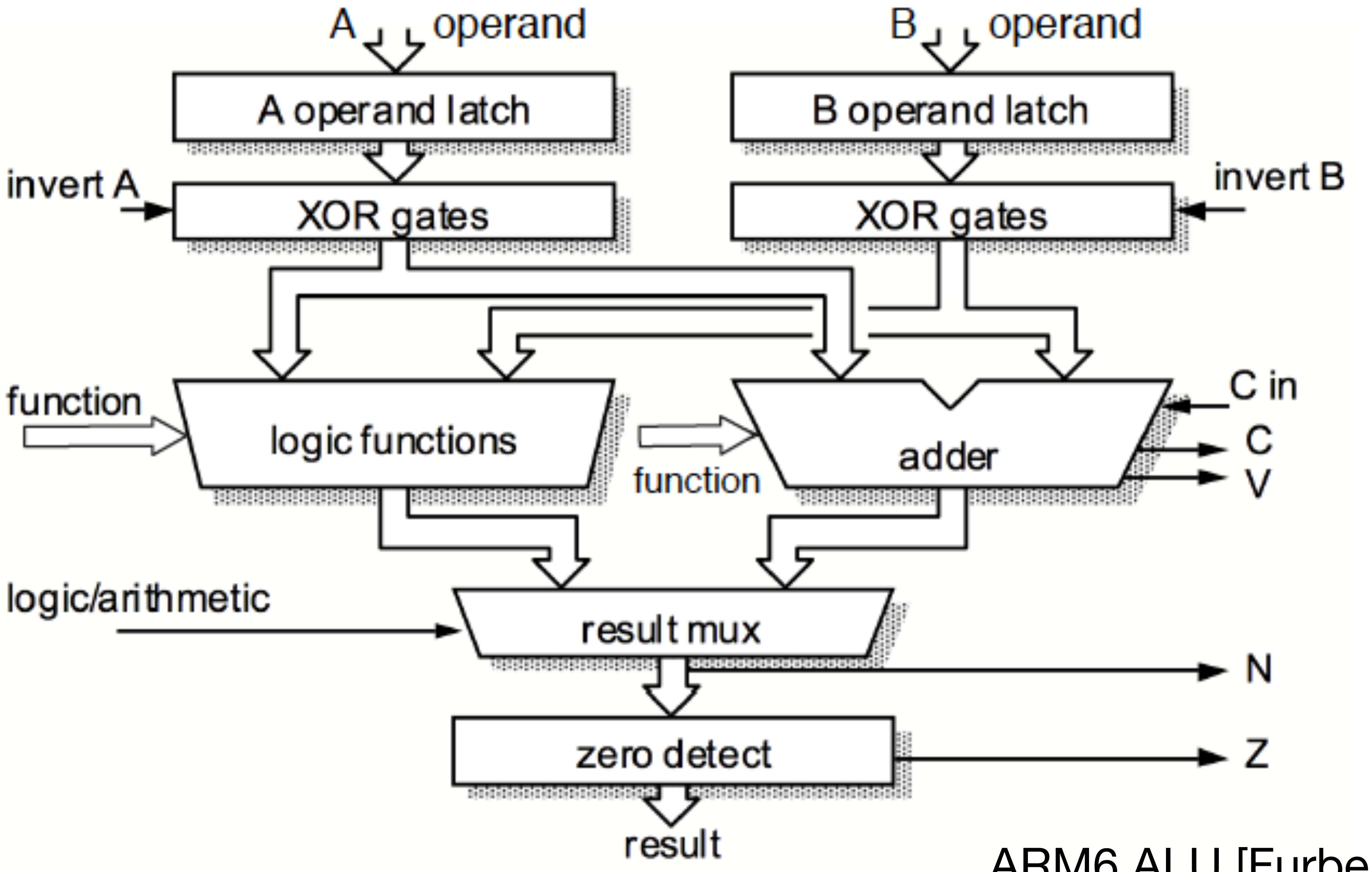
use ALU

test/ compare data & set CCR bits

move data

Data Processing Instructions

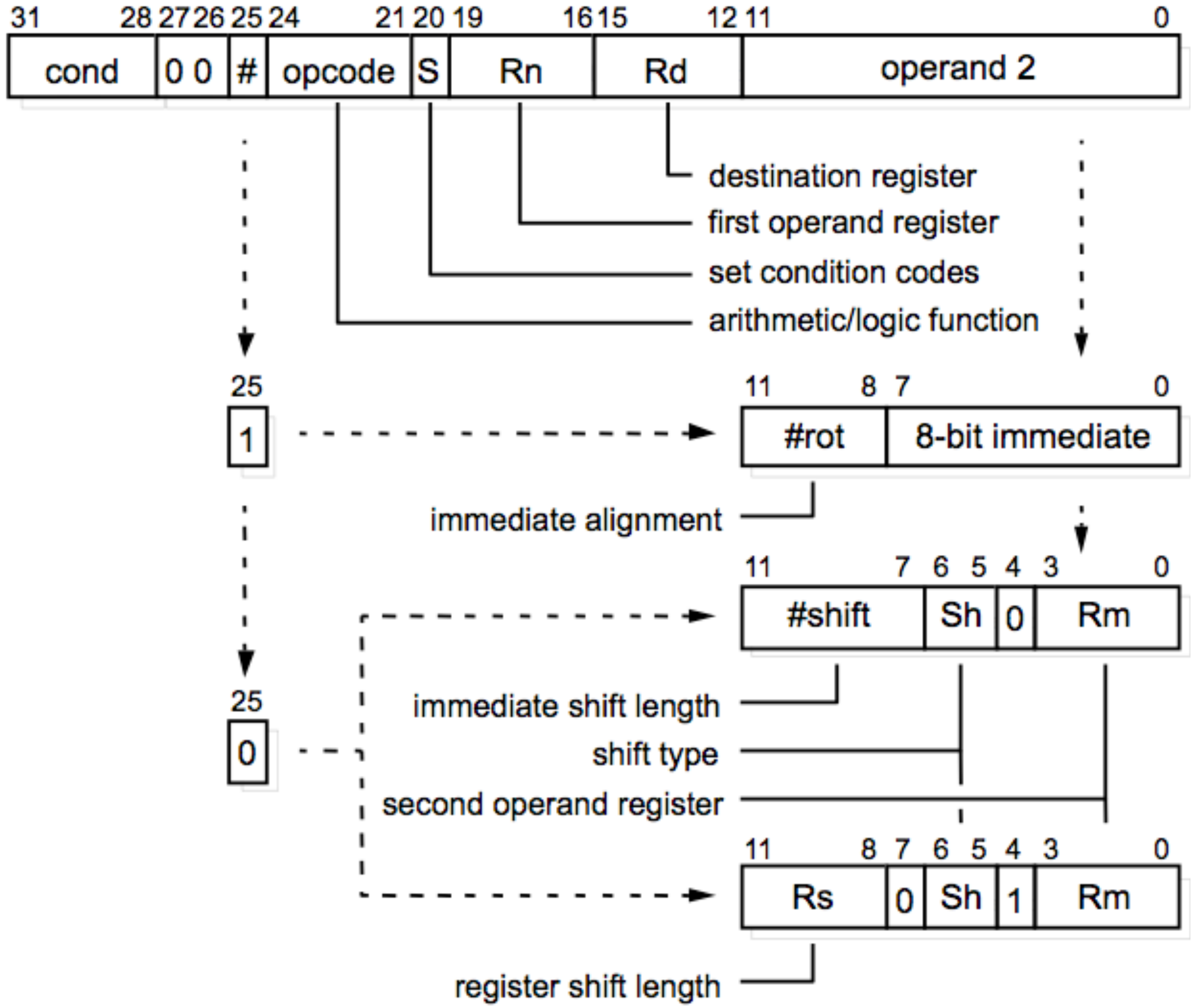
- Execution:



ARM6 ALU [Furber, 2000]

Data Processing Instructions

- Format:



[Furber, 2000]

Shift Operations

- Multiplication can be implemented with shifts, e.g. `r0 × 35`:

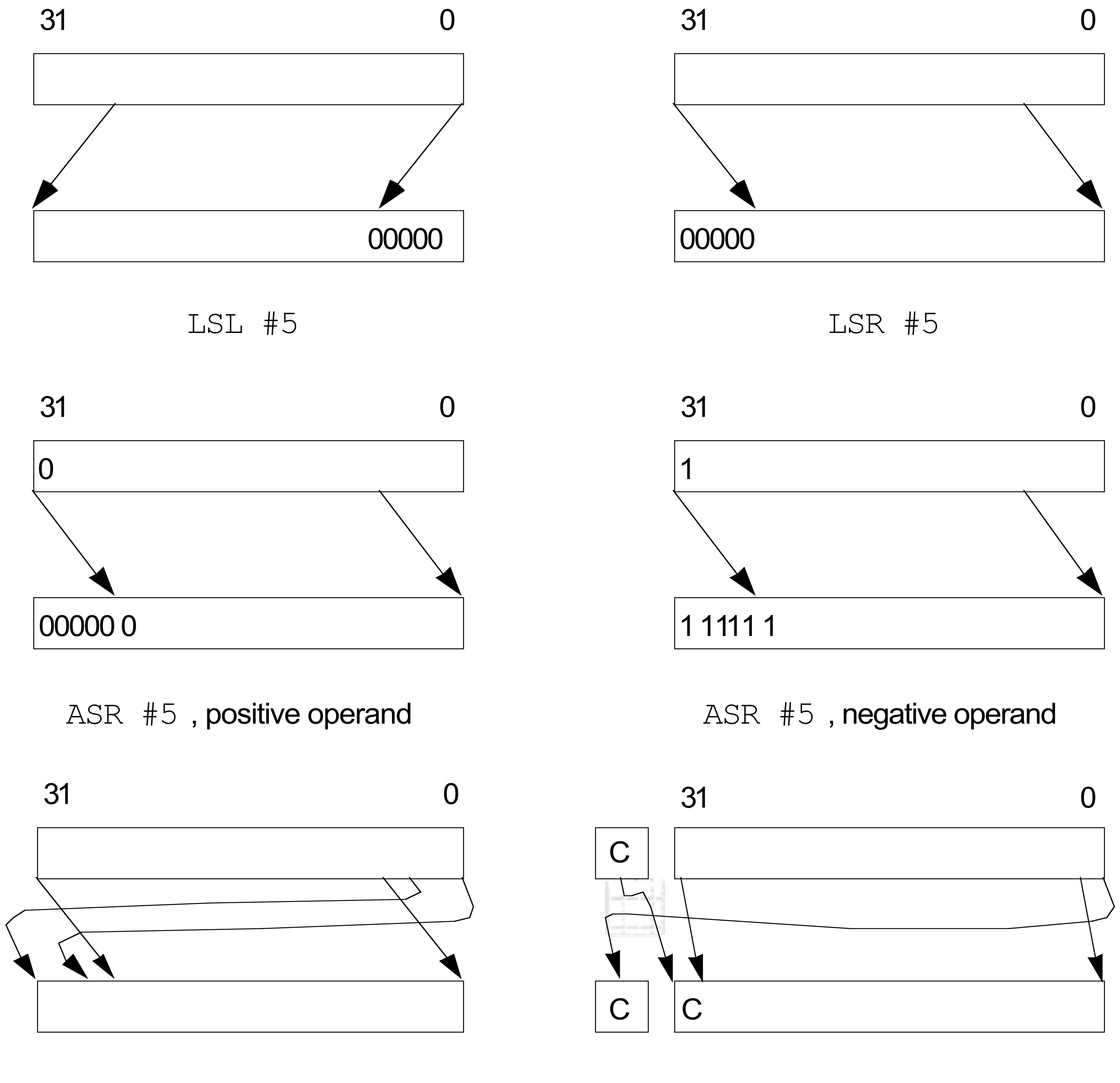
```
ADD r0, r0, r0, LSL #2     ; r0’ = 5 x r0
RSB r0, r0, r0, LSL #3     ; r0’’ = 7 x r0’
```

- Here RSB is “reverse subtract without carry”:

```
RSB Rd, Rn, Operand2    ; Rd ← Operand2 - Rn
```

- In the example `Operand2` is a *shifted register operand:* `r0, LSL #3`

Shift Operations



`LSL #5`

`LSR #5`

`ASR #5` , positive operand

`ASR #5` , negative operand

`ROR #5`

`RRX`

Types of ARM Assembly Instructions

- Data Processing Instructions
- Data Transfer / Movement Instructions
- Control Flow Instructions

Ciaran Moore
ciaran.moore@canterbury.ac.nz

Data Transfer/ Movement Instructions

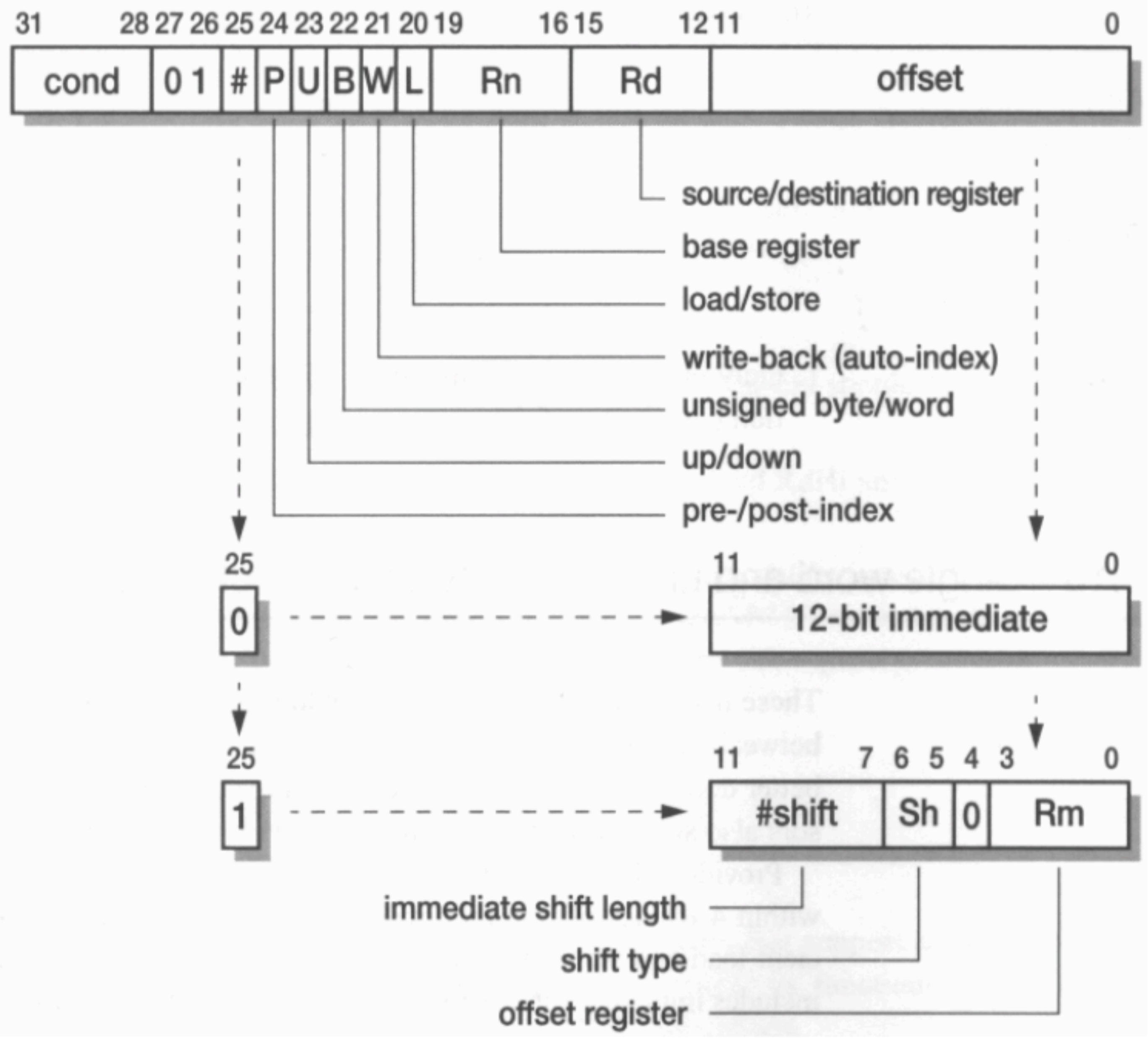
- Can be either:
  - Single register load / store instructions
  - Multiple register load / store instructions
  - Single register swap instructions
- A value is used in a base register and this forms a memory address for data load / store.
  (This is known as *register indirect addressing.*)
- Examples:

```
LDR  r0, [r1]    ; r0 = mem[r1]
STR  r0, [r1]    ; mem[r1] = r0
```

Data Transfer/ Movement Instructions

- Single words or unsigned bytes can be transferred between memory and registers using these instructions.
- Example instructions include load register (`LDR`) and store register (`STR`).
- An effective address typically includes a base register `Rn` plus an `offset`.

Data Transfer/ Movement Instructions



[Furber, 2000]

Data Transfer/ Movement Instructions

- Specialised addressing modes can be used with data transfer instructions.
- Register indirect addressing:
  - Uses a *base plus offset*; the base is a register that can be pre- or post-indexed.
  - The base register can be the source and/or destination operand.
  - Examples:

```
LDR r0, [r1, #4]     ; r0:=mem[r1+4] (pre-index)
LDR r0, [r1, #4]!    ; r0:=mem[r1+4]; r1:=r1+4
                     ;      (pre-index with auto-indexing – !)
LDR r0, [r1], #4     ; r0:=mem[r1]; r1=r1+4 (post)
```

Types of ARM Assembly Instructions

- Data Processing Instructions
- Data Transfer / Movement Instructions
- Control Flow Instructions
    - Branches
    - Comparison Operations
    - Conditional Execution

Control Flow Instructions

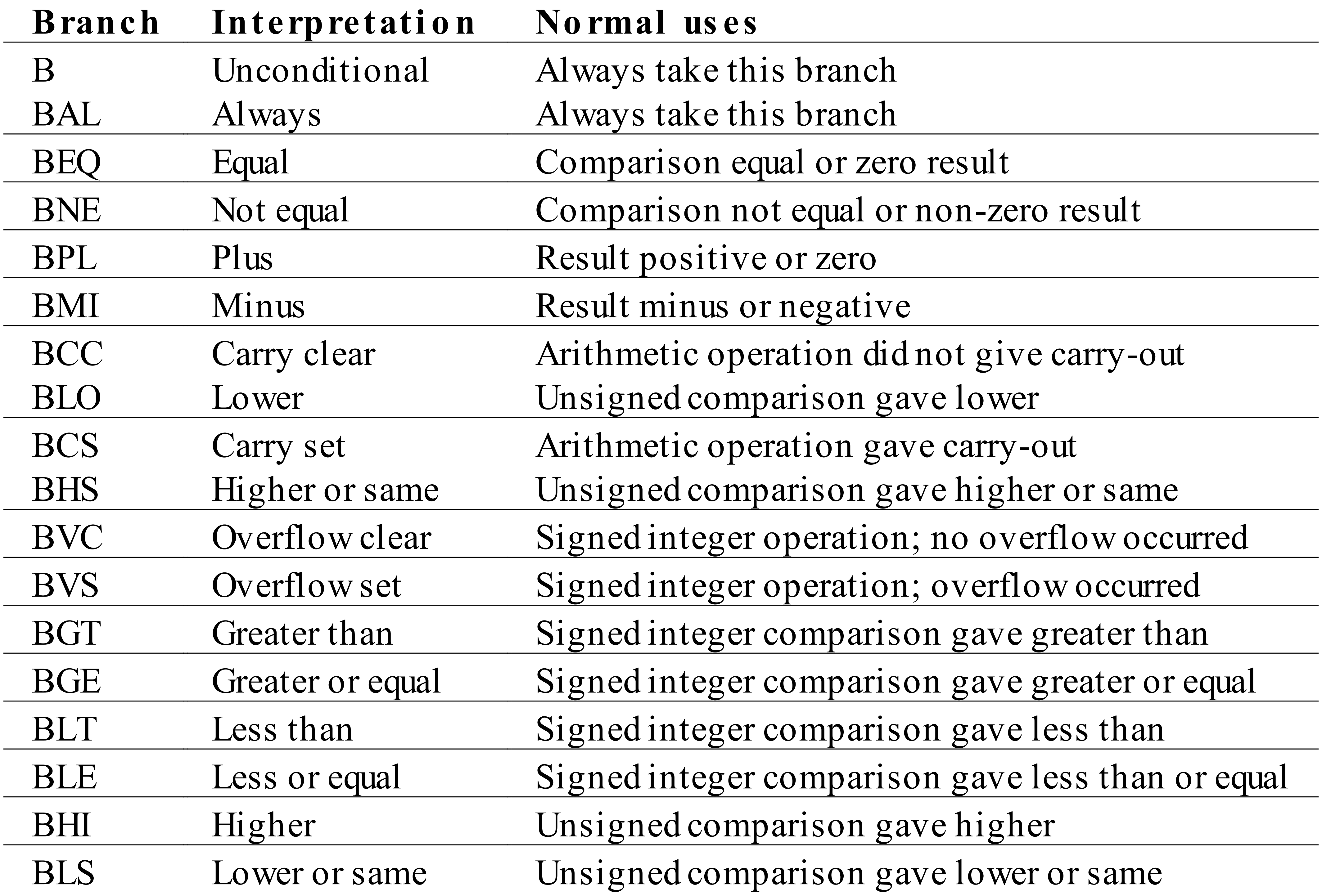
- Determine which instruction executes next.
- Most common is the *branch* (`B`) instruction:

```
Loop        B   Loop
```

- Conditional branches (`BNE`, `BCC`, `BLO`, etc.) test the condition codes in the CCR (e.g. `N`, `V`, `Z`, `C`) to determine if a branch should be taken.

```
            MOV     r0, #0          ; for (i=0; i<10; i++)
LOOP    …                           ; {…};
            ADD     r0, r0, #1
            CMP     r0, #10
            BNE     LOOP
```

Control Flow Instructions



Branch	Interpretation	Normal uses
B	Unconditional	Always take this branch
BAL	Always	Always take this branch
BEQ	Equal	Comparison equal or zero result
BNE	Not equal	Comparison not equal or non-zero result
BPL	Plus	Result positive or zero
BMI	Minus	Result minus or negative
BCC	Carry clear	Arithmetic operation did not give carry-out
BLO	Lower	Unsigned comparison gave lower
BCS	Carry set	Arithmetic operation gave carry-out
BHS	Higher or same	Unsigned comparison gave higher or same
BVC	Overflow clear	Signed integer operation; no overflow occurred
BVS	Overflow set	Signed integer operation; overflow occurred
BGT	Greater than	Signed integer comparison gave greater than
BGE	Greater or equal	Signed integer comparison gave greater or equal
BLT	Less than	Signed integer comparison gave less than
BLE	Less or equal	Signed integer comparison gave less than or equal
BHI	Higher	Unsigned comparison gave higher
BLS	Lower or same	Unsigned comparison gave lower or same

Control Flow Instructions

- Conditional branch instructions work with condition test instructions, e.g. `CMP r0, r1`, which set or clear the respective conditions flags in the CPSR.
- A conditional branch instruction immediately following a test condition instruction, e.g. `CMP`, will change the program flow if the test condition is met.
- 16 conditional branch instructions are available.
- The status flags in the CPSR can optionally be set *after* instruction execution by appending an '`S`' on an instruction mnemonic, e.g.:

```
SUBS  r0, r0, #1          or         SUB  r0, r0, #1
BEQ    ..                            CMP  r0, #0
                                     BEQ  …
```

More on Comparison Operations (Condition Test Instructions)

- From the list of Data Processing Instructions:

```
CMP  r1, r2    ; set cond. codes (cc) on r1 - r2
CMN  r1, r2    ; set cc on r1 + r2
TST  r1, r2    ; set cc on r1 AND r2
TEQ  r1, r2    ; set cc on r1 XOR r2
```

- Only the cc bits in the CPSR are set or cleared by executing these instructions.

31 28	27 8	7 6	5	4 0
N Z C V	unused	I F	T	mode

N = 1, if MSB of (r1 - r2) is '1'; Z = 1 if (r1 - r2) = 0;

C = 1, if r1 & r2 are unsigned and r1 < r2; V = 1 if (r1, r2) are unsigned and r1 < r2

Conditional Execution

- Conditional instruction execution is an advanced feature supported by ARM cores.
- Any instruction (or group of instructions) may be conditionally executed by appending a condition of execution code as a suffix to the instruction mnemonic:

```
START:
   CMP r0, #5
   BEQ NXT        ;  if (r0 != 5) {
   ADD  r1, r1, r0 ;    r1' = r1 + r0
   SUB  r1, r1, r2 ;    r1 = r1' - r2
 NXT:             ;  }
```

```
START:
 CMP      r0, #5      ; if (r0 != 5) {
 ADDNE  r1, r1, r0 ;   r1' = r1 + r0
 SUBNE  r1, r1, r2 ;   r1 = r1' - r2
  …                    ; }
```

Conditional Execution

- Example: cascading multiple conditional statements

```
;; If (x==y) && (m==n) j++
        CMP         r0, r1          ; r0=x, r1=y
        CMPEQ       r2, r3          ; r2=m, r3= n
        ADDEQ       r4, r4, #1      ; j = j + 1
```

- Conditional execution is only efficient if the conditional sequence is three instructions or less.
- Use a branch conditional if your conditional sequence is more that three instructions.
- In the example above it is also possible to have an ELSE condition (to the two condition expressions shown): use xxxNE, where xxx is an instruction mnemonic.

Ciaran Moore
ciaran.moore@canterbury.ac.nz

Conditional Execution

Opcode [31:28]	Mnemonic extension	Interpretation	Status flag state for execution
0000	EQ	Equal / equals zero	Z set
0001	NE	Not equal	Z clear
0010	CS/HS	Carry set / unsigned higher or same	C set
0011	CC/LO	Carry clear / unsigned lower	C clear
0100	MI	Minus / negative	N set
0101	PL	Plus / positive or zero	N clear
0110	VS	Overflow	V set
0111	VC	No overflow	V clear
1000	HI	Unsigned higher	C set and Z clear
1001	LS	Unsigned lower or same	C clear or Z set
1010	GE	Signed greater than or equal	N equals V
1011	LT	Signed less than	N is not equal to V
1100	GT	Signed greater than	Z clear and N equals V
1101	LE	Signed less than or equal	Z set or N is not equal to V
1110	AL	Always	any
1111	NV	Never (do not use!)	none

In-Line Assembly Programming

- This is a fast way of executing assembly instructions in C.
- The compiler passes each statement directly and also translates the parameters directly to the assembler.
- Depending on the compiler you may need to precede each statement with the `volatile` keyword.
- Be careful when using in-line assembly. C has its own way of doing things and writing directly to registers such as r13, r14 or r15 (sp, lr, pc) can easily end badly.

```
  asm(" .arm
       mov r0,r1");     // NOTE: This
// worked on earlier GCC compilers

  asm(" .arm\n"
     " mov r0,r1\n");   // NOTE: This
// works on all versions of GCC.

 OR

  asm(" .arm\n\
       mov r0,r1\n");   // NOTE: This also
// works on all versions of GCC.
```

Note: you must include a space between the first quotation mark and your instruction / pseudo-op.

Summary

- Data Processing Instructions:
  `ADD` and `ADDS`; shift operations

- Data Transfer/Movement Instructions:
  `LDR` and `STR`; register indirect addressing

- Control Flow Instructions:
  Code Condition bits & conditional execution suffixes

- In-Line Assembly Programming

Homework

- The ARM7-TDMI has a banked register file; however, the Cortex M series does not support this. How are fast interrupts supported without bank register switching?
- If complex Load and Store operations take more than one instruction cycle to complete, how are low-latency interrupts maintained using the Cortex-M4?
- Given A = 7, B = 3 and Cin = 1, carry out two's compliment subtraction of A - B by hand.

Homework

- Recommended reading: Furber §3.1-3.5 (pp. 50–73) & §6.11 (pp. 186–187).
- Using conditional execution instructions, write the assembly code for:
  `if ((a != b) || (c >= d)) e++;`
- What is the difference between the `LSL`, `LSR` and `ASR` instructions? What about `ASL`?
- In the in-line assembly example above, what does `.arm` mean and why is it shown inside each `asm(…)` code sequence?

Homework

- The condition code register must reside somewhere in the CPU. For most ISAs, is it located in the data path or the control unit?
- Write an instruction operation in register transfer language that subtracts the contents of register r0 from register r1, with a borrow, and places the results in register r6.
- Considering the four general ISA types, which one is used in the Data Processing Instruction example above?
- The Data Processing Instruction example above does not use data memory. Why not?
- The Glossary slide gives two examples of instruction types that execute in stages. How can a 3-stage instruction (or even a 5-stage instruction) execute in one CPU cycle?

References

- Furber, S., *ARM system-on-chip architecture*, 2nd Ed., Addison-Wesley, 2000.
- Atmel Corporation, *AT91 ARM Thumb-based Microcontrollers Datasheet*, Preliminary, November, 2006.
- Mano & Kime, *Logic and Computer Design Fundamentals*, 2nd. Ed., Prentice Hall, 2001.
- Valvano, *Introduction to the ARM Cortex-M Microcontrollers*, 2017.
- Cockerell, P., *ARM Assembly Language Programming*, Computer Concepts Ltd., www.peter-cockerell.net/aalp/html/frames.html