

Chapter 2

Introduction to the Architecture and Organization of a Computer System

2.1 Introduction of a Stored Program Machine

Objectives

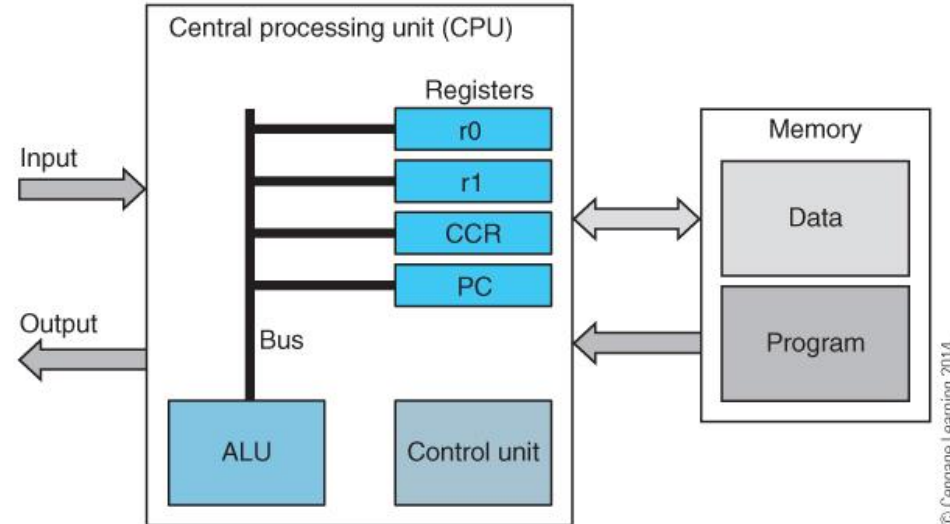
- Before introducing how the ARM microcontroller works, we study a generic stored program machine that has the essential features of an ARM microcontroller, but lacks its complexity
- We will learn about:
 - Components in a stored program machine
 - How each instruction is processed
 - How program flow is established
 - Formats of instructions
 - Brief comparison of RISC and CISC architecture

Components of a computer

1. Memory unit
2. Input/output (I/O) unit
3. Central processing unit (CPU)

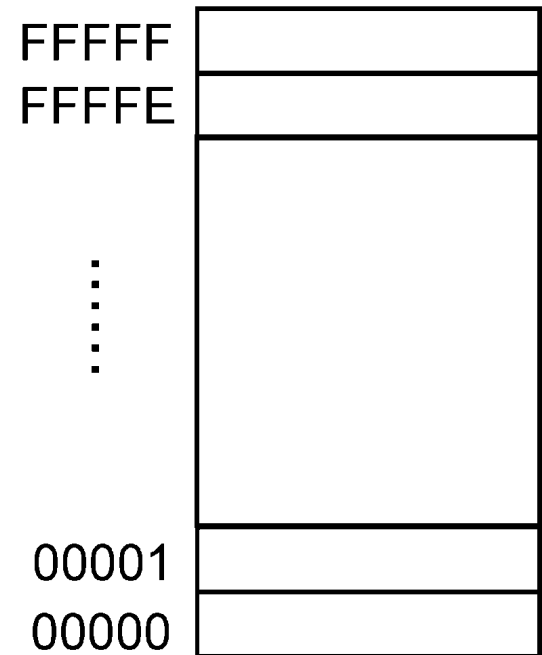
FIGURE 3.1

Fundamental structure of a computer



Memory unit

- Store programs and data.
- Program and data may share the same memory space or they may be separated.
- In ARM, program and data share the same memory
- Arranged sequentially into units.
- Each unit is referred as a memory location and identified by a unique address.
- The larger the number of address bits, the larger the number of memory locations that can be addressed.
- e.g., 20 address bits $\rightarrow 2^{20} = 1,048,576$ locations



Input/Output (I/O) Unit

- Handles the transfer of data between the computer and external devices or peripherals.
- Each unit is identified by a unique address.
- The physical devices used to communicate are called *ports*.
- The equipment for data input/output from outside world is called *peripheral*.
- Typical peripherals are keyboard, video monitor, mouse, printer, disks, tape and CD-ROM drive.

Central Processing Unit (CPU)

- The CPU controls the sequence of operation of the computer.
- Made up of 3 components:
 - Control Unit
 - Registers
 - Arithmetic-Logic Unit (ALU)

Control Unit

- Controls the processing of an instruction
- Brain of the brain
- The control unit consists of circuitry for:
 - *Fetching*
 - *Decoding* an instruction then using the decoded information to control data flow between registers and ALU

Registers in CPU

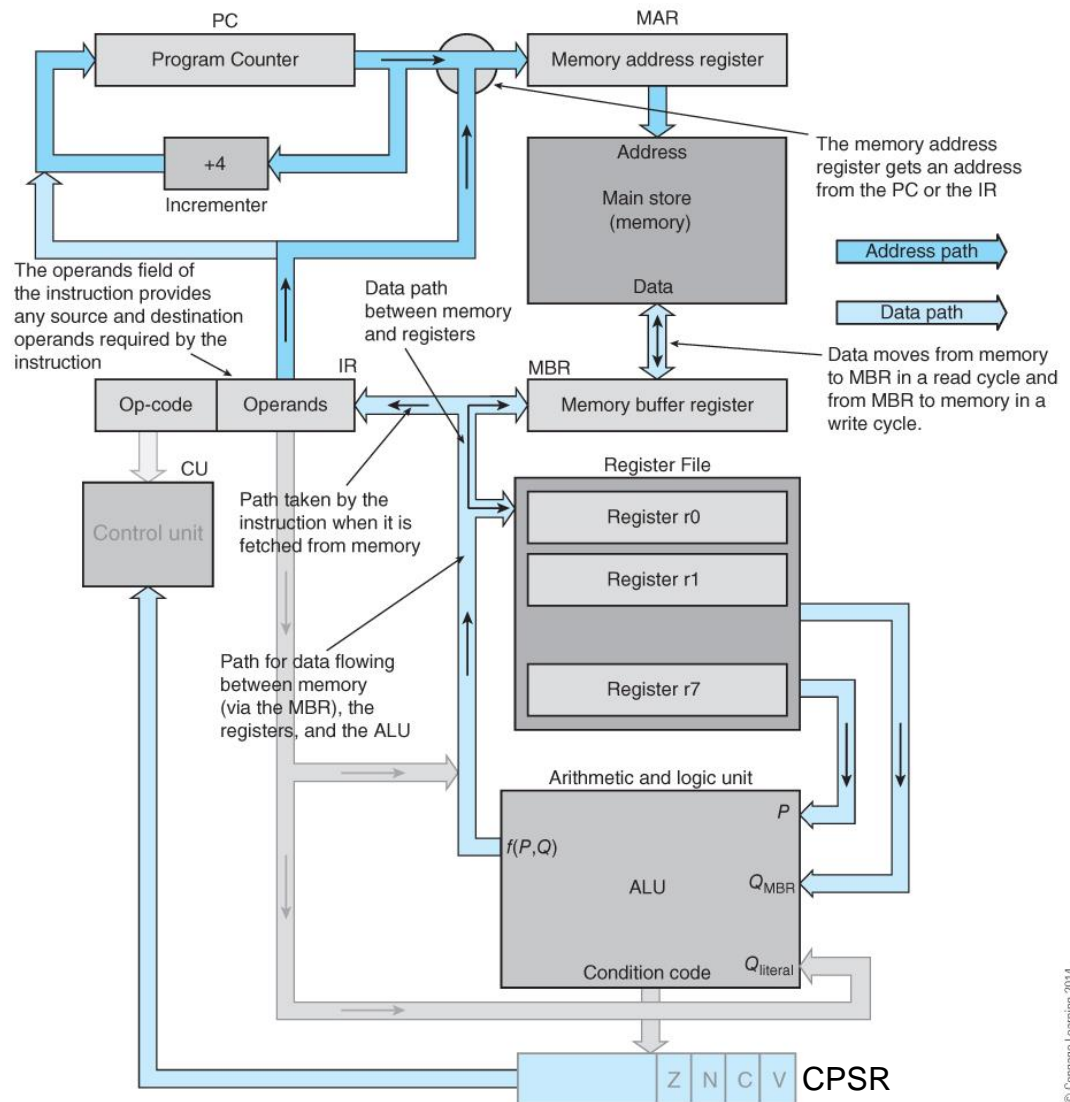
- Temporary storage within a processor.
- Not all registers can be accessed by users.
- Some are special-purpose and some are general-purpose.

Registers in our simple computer

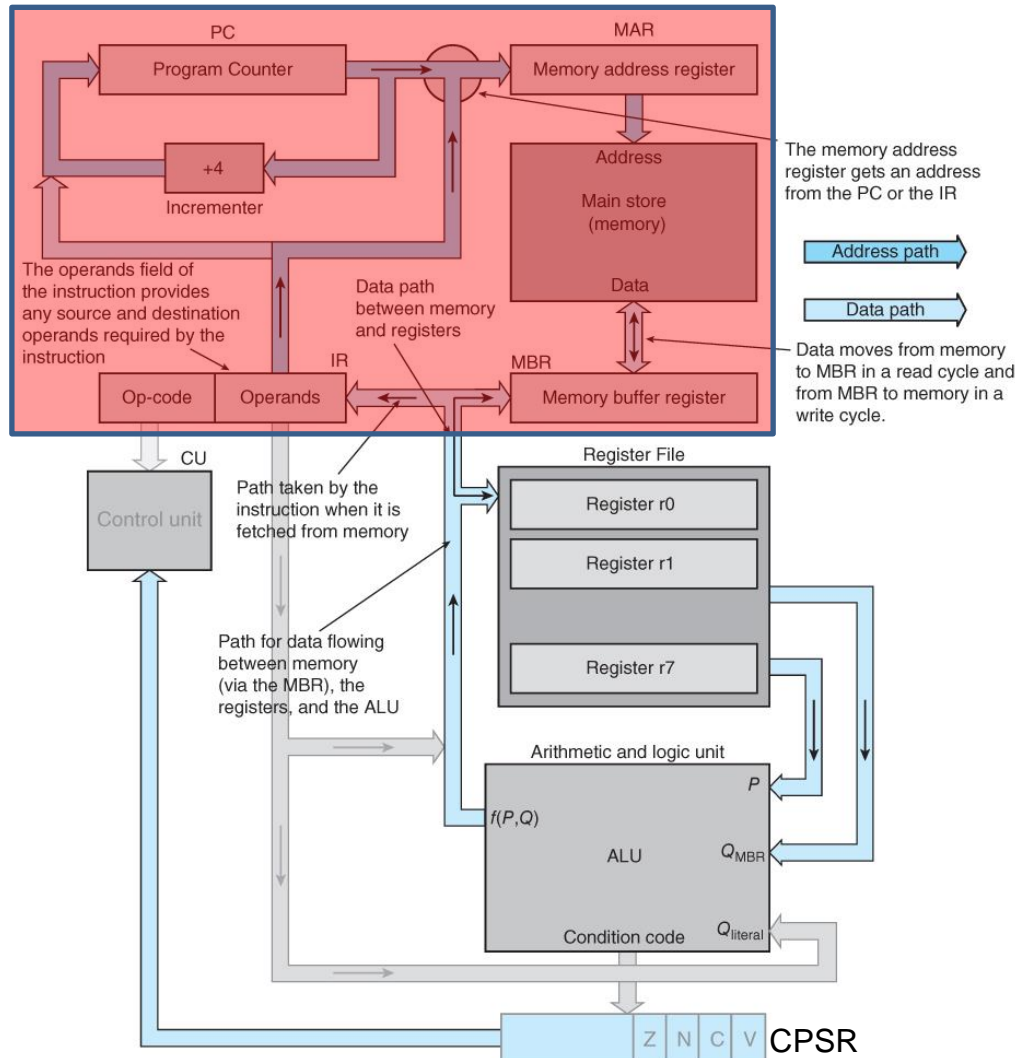
- **PC:** The *program counter* contains the address of the next instruction to be executed. Thus, the PC points to the location in memory that holds the next instruction.
- **IR:** The *instruction register* holds the instruction that is currently being executed. The instruction identifies the operation (e.g., ADD) and specifies the *address of operands*.
- **r0 – r7:** The register file is a set of eight general-purpose registers r0, r1, r2, ..., r7 that store temporary (working) data, for example, the intermediate results of calculations.
- **CPSR:** The *current program status register* stores a set of one-bit flags the processor sets or clears during the execution of each instruction.
- **MAR:** The *memory address register* stores the address of the location in main memory that is currently being accessed by a read or write operation.
- **MBR:** The *memory buffer register* stores data that has just been read from main memory, or data to be immediately written to main memory.

Structure of our simple computer

- Instruction processing consists of two phases:
 - Fetch
 - Execution



Instruction Processing: Fetching



1. PC supplies an address to the memory address register (MAR), which holds it while the instruction is looked up in memory.

2. The address in PC is incremented by 4 so that it points to the next instruction in the program.

3. The instruction is loaded into the memory buffer register, MBR, and then copied to the instruction register, IR where the op-code is decoded.

RTL Notation

- Register Transfer Language (RTL) Notation is a *notation* used to define operations used heavily in this course.
- Square brackets indicate the contents of a memory location. For example:
 - $[20] = 5$ states that the contents of memory location 20 are equal to the number 5.
- The arrow symbol, \leftarrow , indicates a data transfer. For example:
 - $[20] \leftarrow 6$ states that the number 6 is put into memory location 20.
 - $[20] \leftarrow [6]$ indicates that the contents of location 6 are copied into location 20.

Fetching in RTL

$[MAR] \leftarrow [PC]$;copy PC to MAR

$[PC] \leftarrow [PC] + 4$;increment PC

$[MBR] \leftarrow [[MAR]]$;read instruction pointed at by MAR

$[IR] \leftarrow [MBR]$;copy instruction in MBR to IR

Instruction Processing: Execution

Before discussing execution, there is a need to distinguish between two types of instructions in our computer:

a. Arithmetic and logic operations with format:

Operation register_{destination}, register_{source1}, register_{source2}

b. Load (LDR) and store (STR) operations with formats:

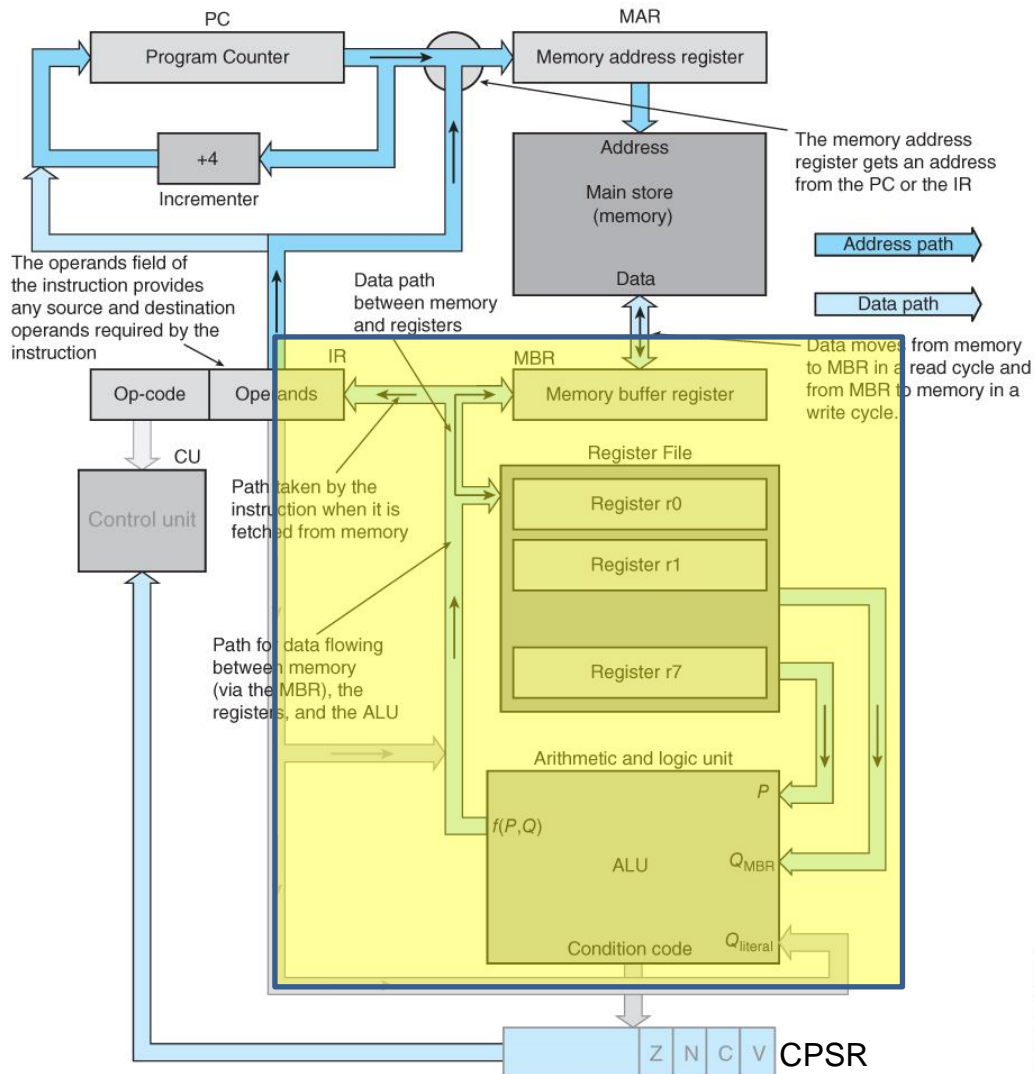
LDR register_{destination}, memory_{source}

STR register_{source}, memory_{destination}

Note: LDR: memory → register

STR: register → memory

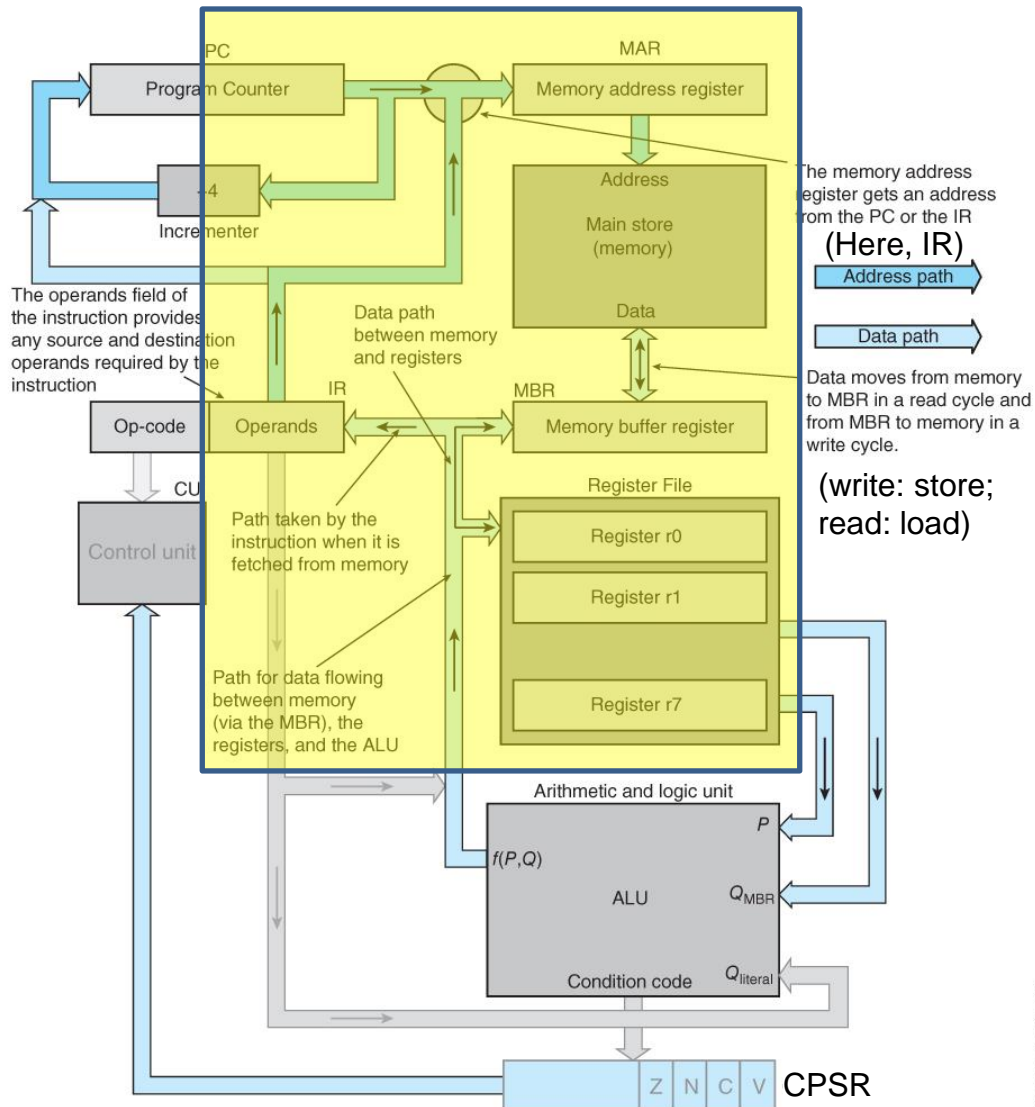
Instruction Processing: Execution



1. The control unit decodes an instruction.

2a. For arithmetic and logical instructions, the operands in the register file are transferred to the ALU where they are operated on and then the result passed to the destination register.

Instruction Processing: Execution



2b. For load and store operations, the memory address in the instruction register is sent to the MAR for a read or write operation to be performed.

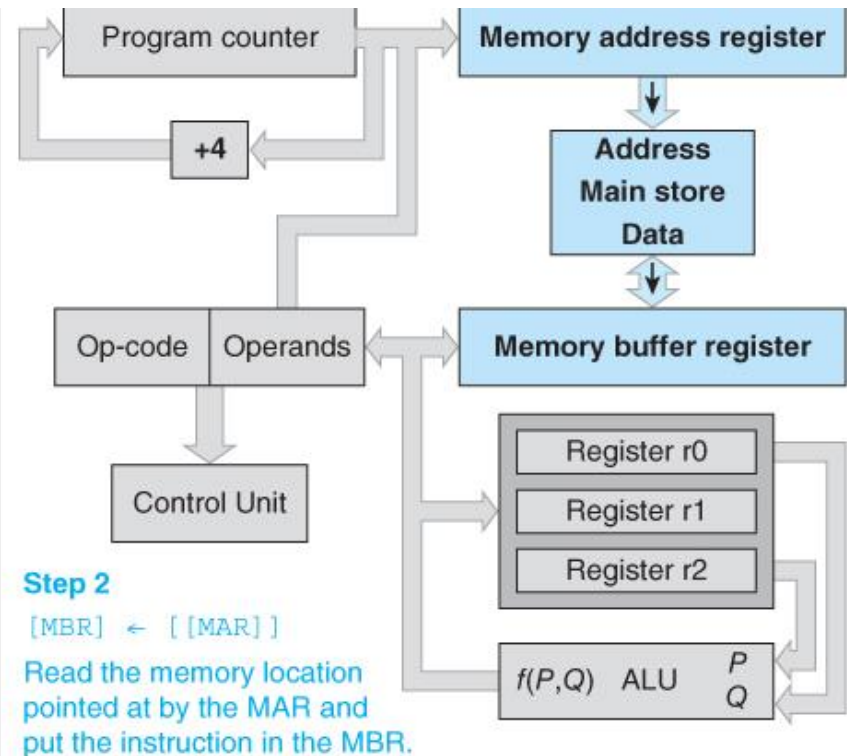
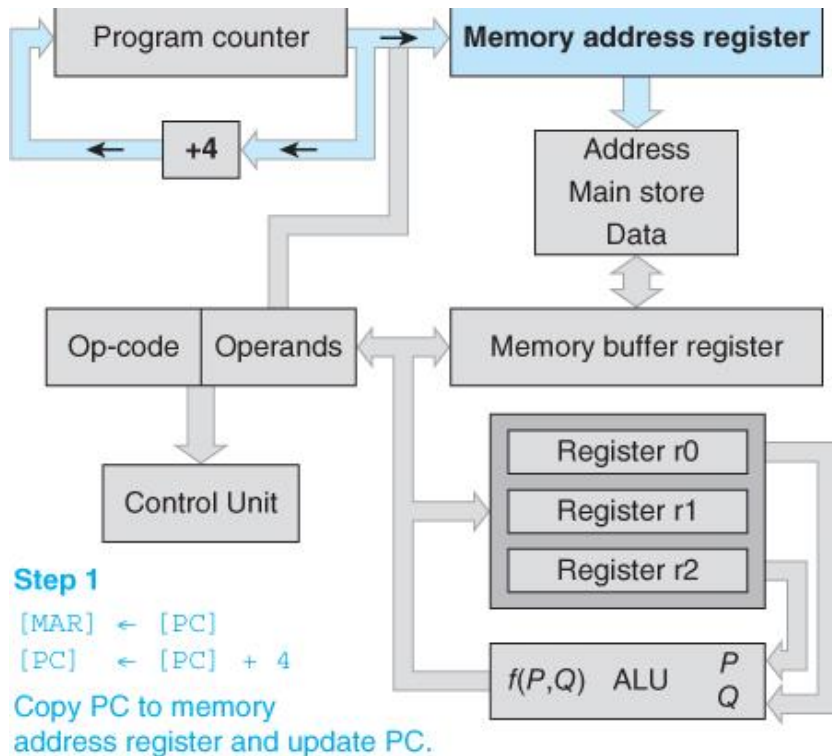
Executing LDR in RTL

$[MAR] \leftarrow [IR(address)]$;copy operand address from IR to MAR

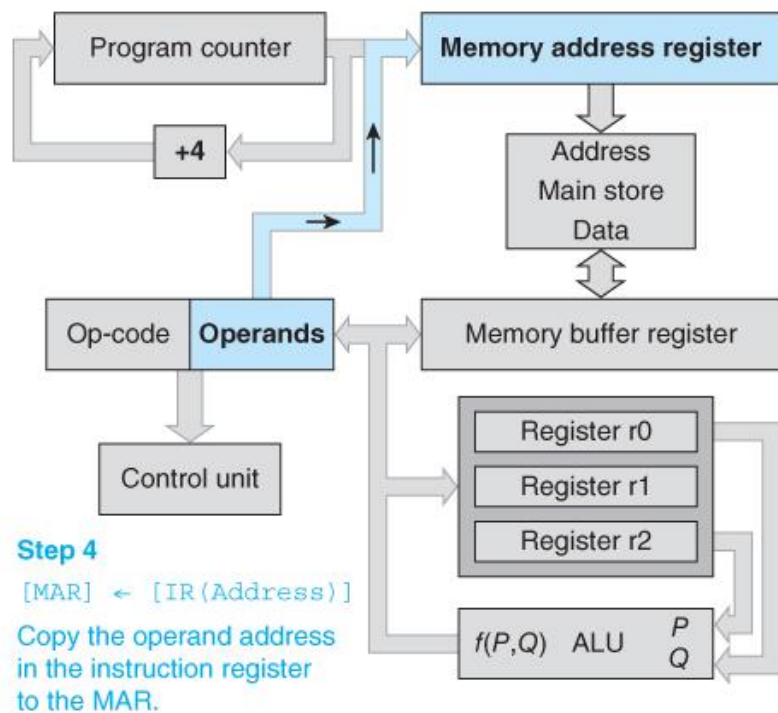
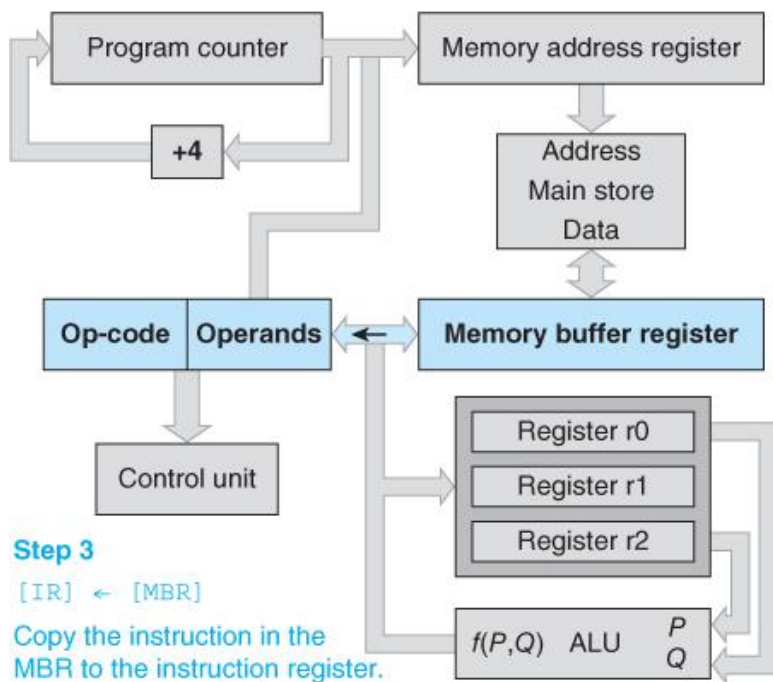
$[MBR] \leftarrow [[MAR]]$;read operand value from memory

$[r1] \leftarrow [MBR]$;add the operand to register r1

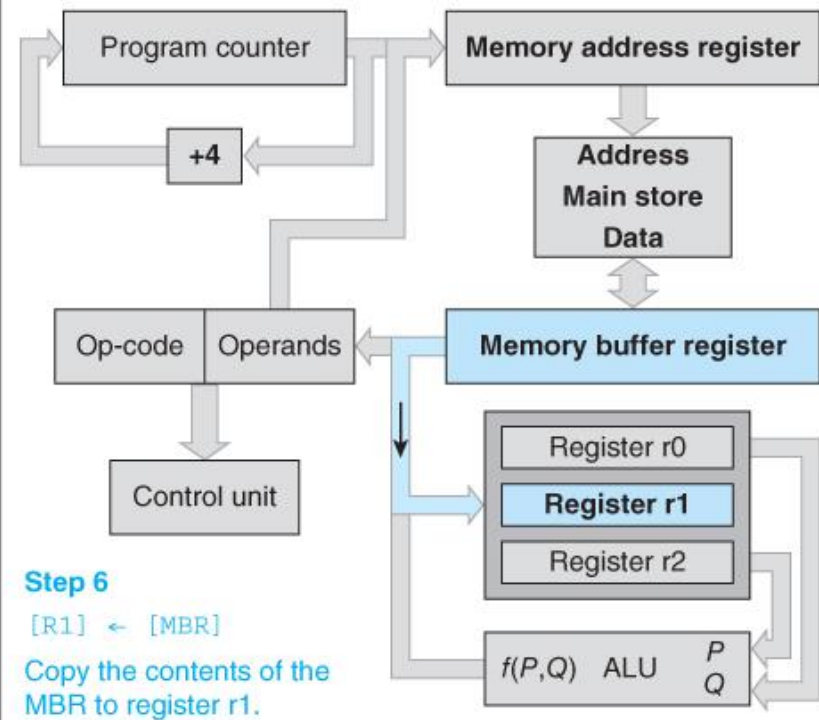
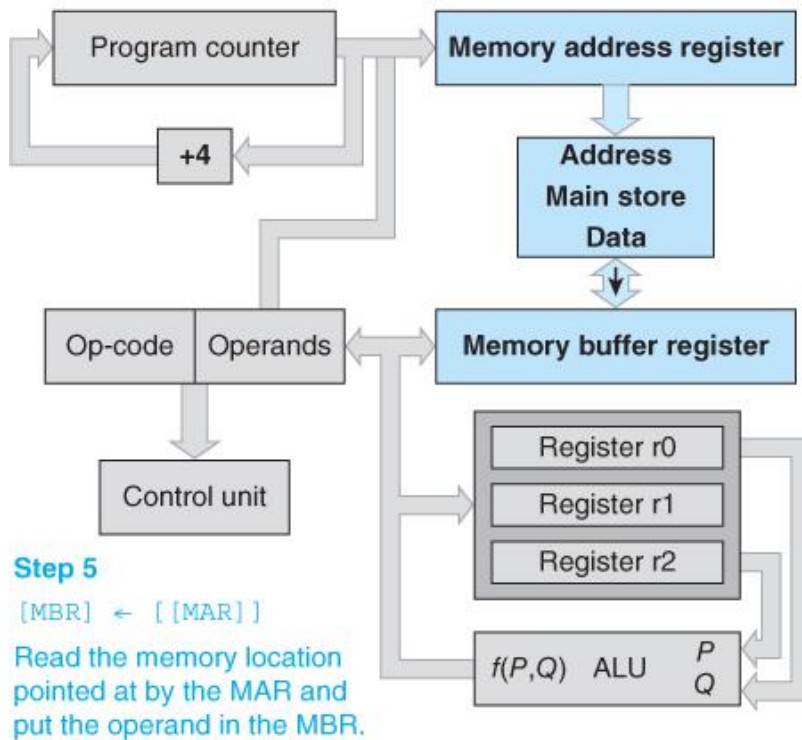
Putting it together (1)



Putting it together (2)



Putting it together (3)

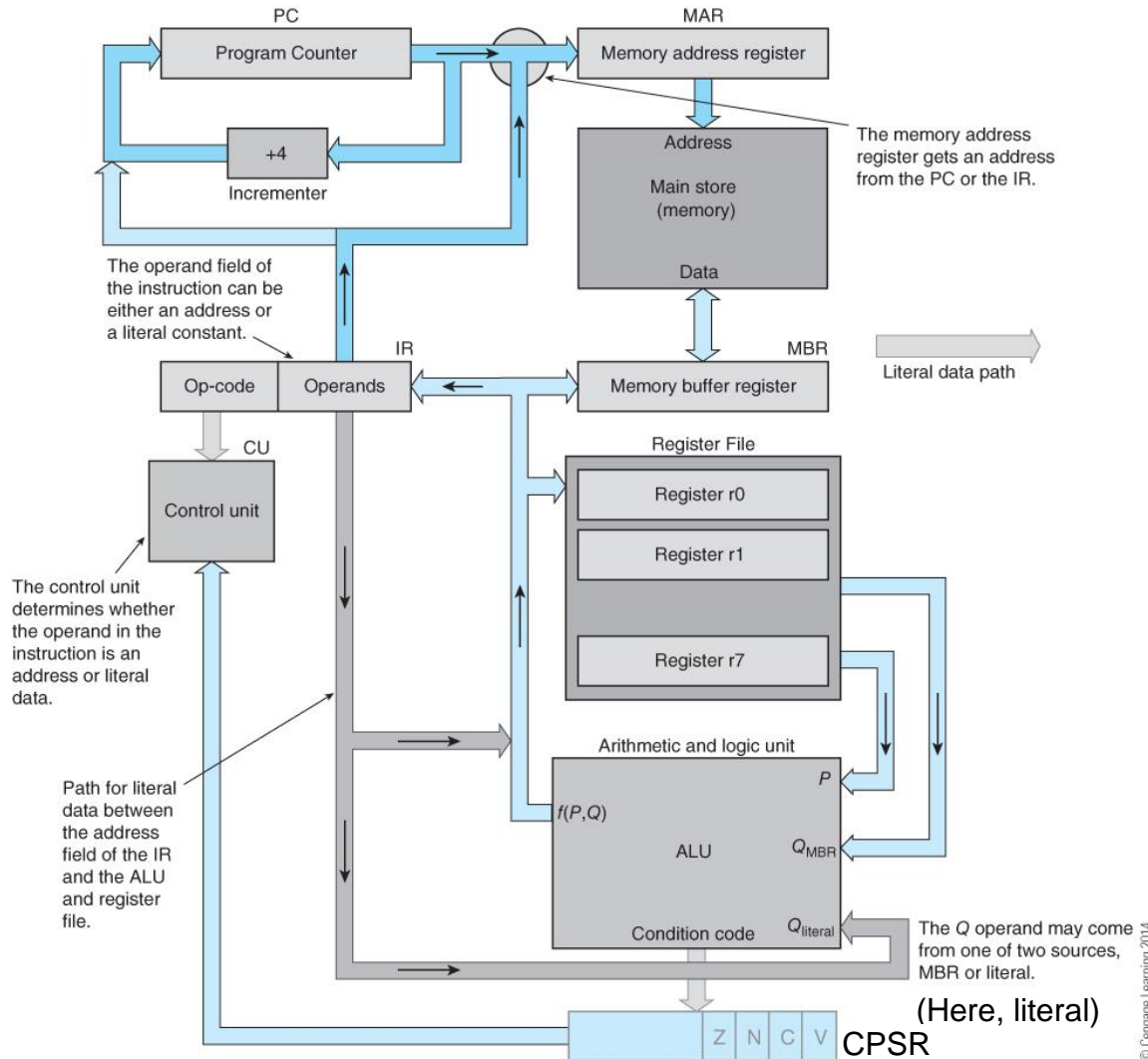


Extension: Dealing with constant

- So far, we have worked with *the contents of a memory location*.
- Suppose we want to
 - load the *number 25 itself* into register r1, or
 - add the value 25 to contents of r1 and puts sum in r0
- The number 25 here is called a *literal* operand.
- A literal operand is prefixed by a hash (#) symbol:
 - LDR r1, #25
 - ADD r0, r1, #25

Extension: Dealing with constant

FIGURE 3.4 Information paths for literal operands



To execute `ADD r0,r1,#25`, the operand to be added with r1 is routed from the operand field of the IR, rather than from the memory system

Extension: Flow control

- *Flow control* refers to any action that modifies the strict instruction-by-instruction sequence of a program.
- *Conditional behavior* allows a processor to select one of two possible courses of action.
- A *conditional instruction* like BEQ results in either continuing program execution normally, or loading the PC with a new value and executing a *branch* to another region of code.
- Branching decision is made based on flags in the CPSR.

CPSR

- When the computer performs an operation, it stores the *status* in the CPSR.
- The processor records whether the result is zero (Z), negative in two's complement terms (N), generated a carry (C), or arithmetic overflow (V).
- N and V are applicable only for signed arithmetic.
- $N = 1$ if the sign bit (the most significant bit) equals to 1

Review: Rules for determining V

- -ve number + +ve number
→ $V = 0$
- +ve number + +ve number
 - OV = 0 if result is +ve
 - OV = 1 if result is –ve
- -ve number + -ve number
 - OV = 0 if result is –ve
 - OV = 1 if result is +ve

Example on CPSR flags

- **The microcontroller does not know whether you are performing a signed or an unsigned addition. The result is the same either way.**
- You can interpret an addition operation as a signed or unsigned operation.
- e.g., 82+22 always gives a result A4
 - Unsigned: $82+22 = A4$
 - Signed: $-7E(\text{represented as } 82)+22 = -5C(\text{represented as } A4)$. V and N only make sense when an operation is interpreted as a signed operation.

$$\begin{array}{r} 82 \\ +22 \\ \hline A4 \end{array}$$

Z=0, C=0
N=1, V=0

Further examples

Example 1

$$\begin{array}{r} 33 \\ +42 \\ \hline 75 \end{array}$$

Z=0, N=0
C=0, V=0

Example 2

$$\begin{array}{r} FF \\ +01 \\ \hline 400 \end{array}$$

Z=1, N=0
C=1, V=0

Example 3

$$\begin{array}{r} DC \\ +C1 \\ \hline 49D \end{array}$$

Z=0, N=1
C=1, V=0

Example 4

$$\begin{array}{r} 5C \\ +41 \\ \hline 9D \end{array}$$

Z=0, N=1
C=0, V=1

An operation depending on Z

```
        SUBS  r5,r5,#1    ;Subtract 1 from r5
        BEQ   onZero      ;IF zero then go to the line labeled 'onZero'
notZero  ADD   r1,r2,r3    ;ELSE continue from here
        .
onZero   SUB   r1,r2,r3    ;Here's where we end up if we take the branch
```

- **SUBS r5,r5,#1** subtracts 1 from the contents of register r5. *If the content of r5 is 0, the Z flag in the CPSR is set; otherwise $Z = 0$.*
- *Note: ARM requires the programmer to update the status flags in CPSR. This is done by appending an “S” to an instruction (e.g., SUBS here)*
- **BEQ onZero** forces a branch to the line labeled ‘onZero’ if the outcome of the last operation was zero (i.e., if $Z = 1$)
- Otherwise, the next instruction in sequence after the BEQ is executed.
- This implements: if zero then $r1 = r2 + r3$ else $r1 = r2 - r3$.


An operation depending on C

- Suppose you have a computer capable of adding 8-bit values but you want to add two 16-bit values.
- You can divide the 16-bit addition to two 8-bit additions, but the answer is not correct if there is a carry in the addition of the least significant byte:

Addition of two 8-bit number
without carry

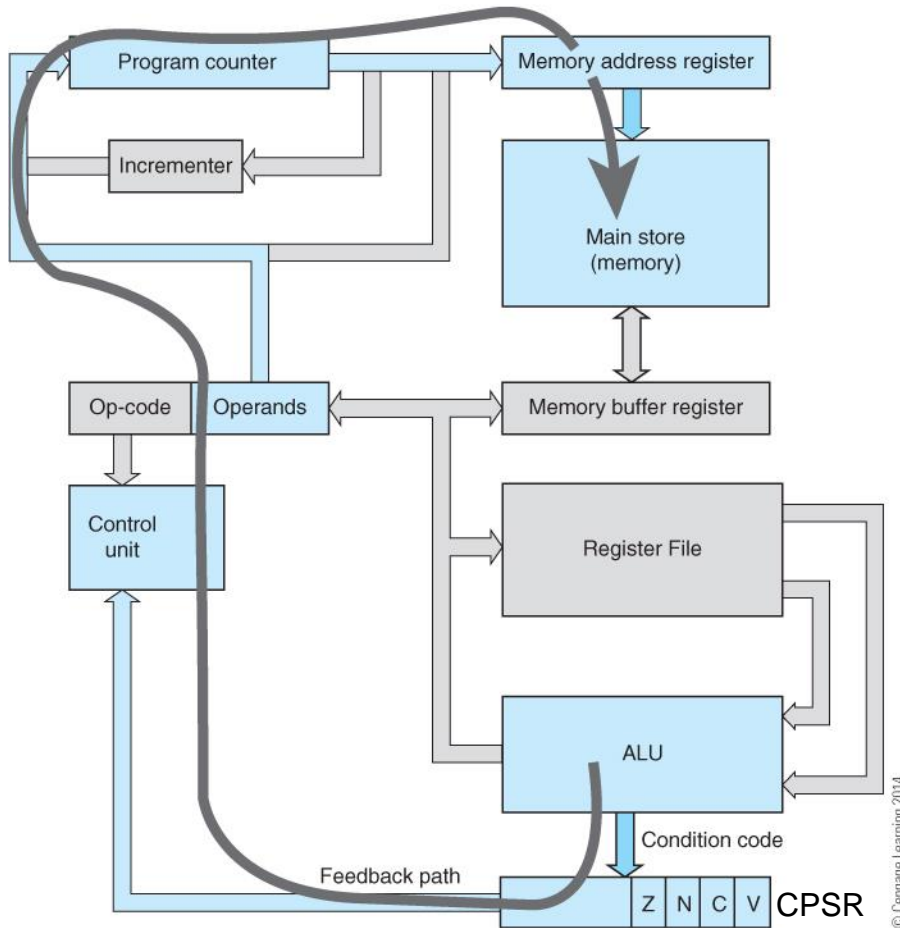
$$\begin{array}{r} \text{34} \quad \text{32} \\ + \text{57} \quad \text{DF} \\ \hline \text{8B} \quad \text{11} \end{array}$$

Addition of two 8-bit number
with carry

$$\begin{array}{r} \text{34} \quad \text{32} \\ + \text{57} \quad \text{DF} \\ \hline \text{8C} \quad \text{11} \end{array}$$


Extension: Flow control

FIGURE 3.6 Feedback from ALU to instruction



```
SUBS r5,r5,#1
BEQ  onZero
notZero ADD r1,r2,r3
.
onZero SUB r1,r2,r3
```

- If $Z = 0$ in CPSR, [PC] does not change and the notZero instruction is executed.
- If $Z = 1$ in CPSR, the PC is loaded with a new address from the operand field of the instruction (i.e., onZero) → fetch/execute the onZero instruction

2.2 Component of Instruction Set Architecture

Instruction Set Architecture

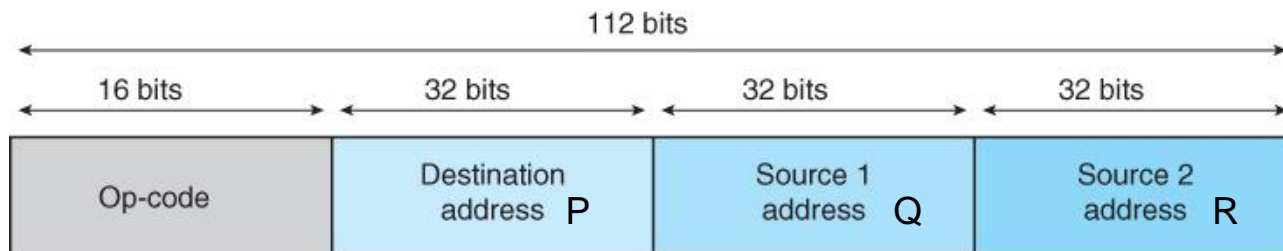
- Three components of an instruction set architecture:
 - Registers
 - Addressing Modes
 - Instruction Formats

Registers

- Do we need registers in CPU?
- Could we just do arithmetic directly on memory locations?

Computer 1

- No on-chip register
- Address of a memory location is 32 bits.
- How many bits do we need to encode the addition operation $P = Q + R$?

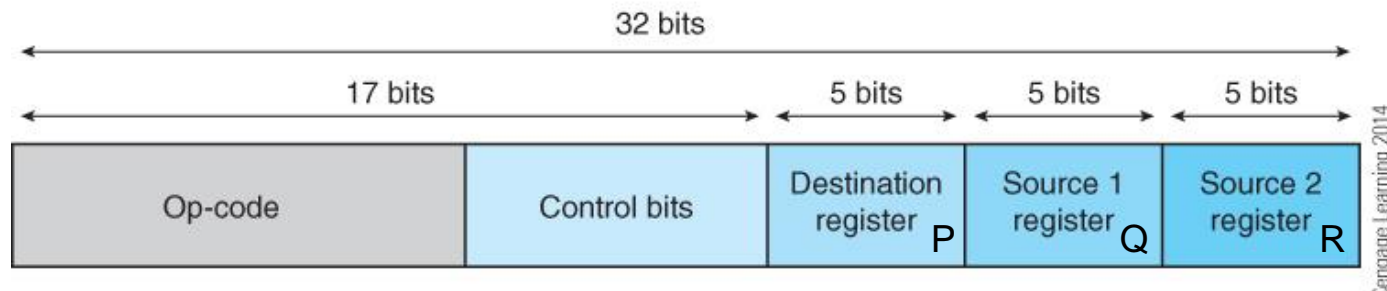


- Not feasible. Instructions in a typical computer is only 16 or 32 bits wide.

Registers

Computer 2

- 32 on-chip registers (can be addressed using 5 bits $2^5 = 32$)
- How many bits do we need to encode the addition operation $P = Q + R$?

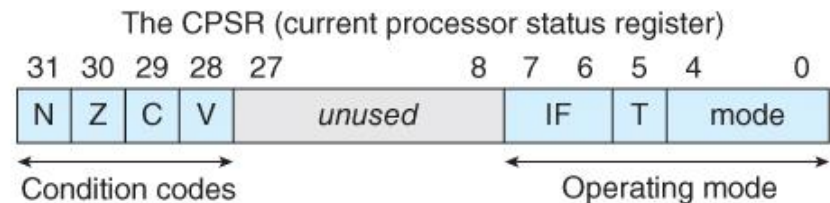
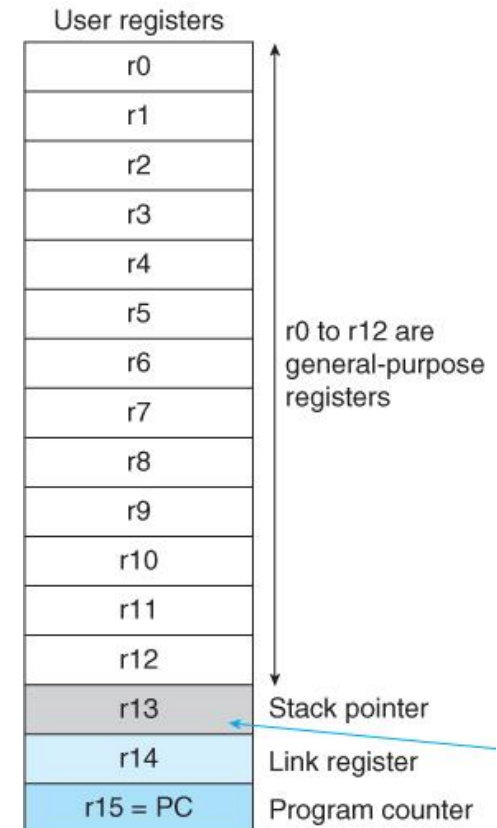


Registers

- Registers are usually the same width as the fundamental word of a computer
- The ARM processor has 32-bit registers and its basic wordlength is 32 bits wide.
- All instructions are 32-bit.

ARM registers

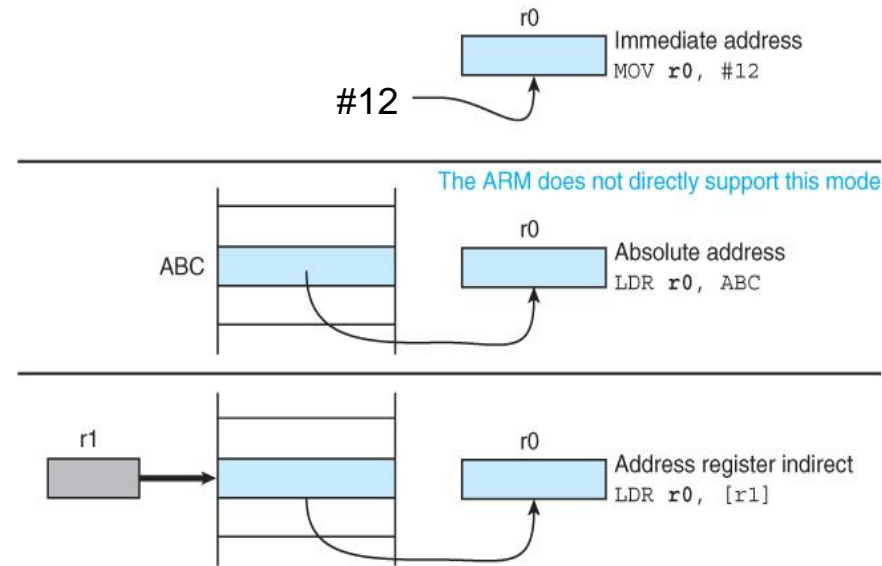
- 16 program-visible registers r0 to r15:
 - Need 4 bits to identify a register in an instruction ($2^4 = 16$)
 - r0-r12: interchangeable. Free to use in whatever way you want.
 - r13: stack pointer (covered later)
 - r14: link register that holds a subroutine return address (covered later)
 - r15: program counter.
- A status register called current program status register (CPSR):
 - Z (zero), N (negative), C (carry) and V (overflow)
 - Some operating mode information involved in interrupt handling mechanism (discussed later)



Addressing Mode

Three fundamental addressing modes:

- *Literal or immediate*: the actual value is part of the instruction
 - e.g., `ADD r1, r2, #5` performs $[r1] \leftarrow [r2] + 5$
- *Direct or absolute*: the instruction provides the memory address of the operand
 - Not implemented in ARM
- *Register indirect*: a register contains the address of the operand
 - e.g., `LDR r0, [r1]` copies to r0 the content of the memory location with address stored in r1.
 - Useful in access tables and arrays, as register indirect addressing can be done with displacement:
 - e.g., `LDR r0, [r1, #2]` copies to r0 the content of the memory location with $[r1]+2$.



Instruction Formats

- There is no fundamental differences between registers and memory locations.
- But accessing register is faster and encoding in an instruction the address of registers require fewer bits.
- As such, the destination and source(s) of operations in ARM are registers:
 - Operation register_{destination}, register_{source1}, register_{source2}
 - e.g., ADD r1, r2, r3; [r1] = [r2]+[r3]
- Data are loaded (LDR) before and stored (STR) after an operation:
 - LDR register_{destination}, memory_{source}
 - STR register_{source}, memory_{destination}
 - Note: LDR: memory → register
STR: register → memory

RISC vs. CISC

- ARM uses a Reduced Instruction Set Computer (RISC) architecture.
- RISC is in contrast with the Complex Instruction Set Computer (CISC)
- RISC utilizes a small, highly-optimized set of instructions.
- CISC implements complex instruction sets in which a single instruction will perform loading, evaluating and storing operations.

Merits of RISC architecture

1. RISC processor has a fixed instruction size.
 - All instructions in an ARM processor is 4-byte (a word)
 - CISC processors such as x86 have instructions that are 1, 2, 3 or even 5 bytes. Variable instruction size makes instruction decoding difficult.

Merits of RISC architecture

2. RISC processor has a large number of registers
 - Avoid the need for storing temporary data in memory stack
 - Accessing data in memory is much slower than CPU register access
 - More CPU register leads to easier encoding of instructions

Merits of RISC architecture

3. RISC uses load/store architecture

- RISC instructions only load from external memory into CPU registers or store register contents in external memory locations.
- No direct arithmetic/logic operations between a register and an external memory location.
- Programs must first load operands from memory locations to registers, then perform arithmetic/logic operations, and then send the result back to memory.
- Avoid delay in accessing external memory during the execution of arithmetic/logic operations.

Merits of RISC architecture

4. RISC processors have a smaller instruction set
 - The reduced complexity requires fewer transistors, thereby leading to less power consumption.
 - Important for mobile devices, such as cellphone and tablets.