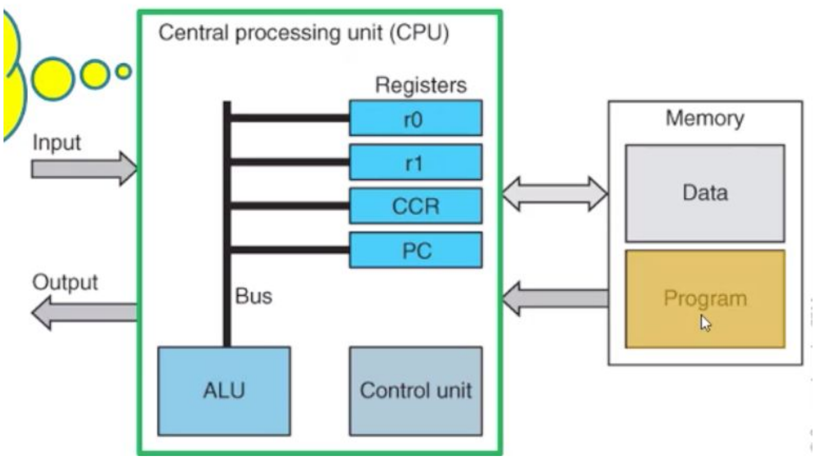
Week 7

Stored Program Computer

- No way to differentiate between instruction set and memory
- We read instructions from memory and send them to the CPU where they are executed
- Operates in 2 different phases: fetch instruction from memory and execute instruction
  - Fetch Phase → next instruction is read from memory (Memory to CPU; CPU is idle)
  - Execute Phase → instruction interpreted and executed by CPU (Memory is idle)
  - Modern computers are pipelined, where fetch and execute operations overlap so CPU and memory are not idle; while CPU is executing, memory fetches next instruction

Stored Program Computer Components:

- Temporary data stored in general-purpose registers (r0, r1, …., ri)
  - Generally, computers have 8, 16, or 32 general-purpose registers
  - Computer requires at least one general-purpose register
- PC (program counter) → register that points to the next instruction to be executed
  - Acts as pointer; holds address (memory location) of next instruction
- IR (instruction register) → stores instruction most recently read from memory; this is the instruction that is currently being executed
  - Once you read the instruction from memory, send to IR where it will start to decode the instruction
- MAR (memory address register) → store an address for a location inside memory; can be used to read/write from memory
- MBR (memory buffer register) → location to store data that you just read from memory or the data you are trying to write to memory
- CCR (conditional code register) → collect conditions that happened out of the ALU (arithmetic logic unit) ie. the flag bits



Instruction Format

- Each processor has its own machine language (consisting of instructions)
- Instruction format defines:
  - # of operands
  - # of bits for each operation
  - Format of each operation
- Computer executes instructions from 8 bits wide to multi-bytes wide
- General format for assembly instructions: Operation registerDestination,registerSource1,registerSource2

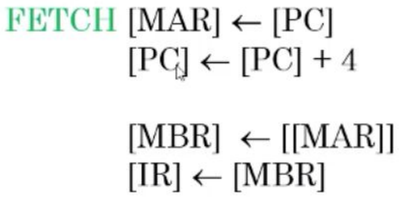
Flow of Stored Program Computer

Fetch Phase:

- In fetch phase, the PC supplies the address of the next instruction to the MAR and the PC is incremented by the size of an instruction in bytes 8. (to read the next instruction)
- Instruction is read into MBR and then copied into IR, where it is decoded

PC → MAR (PC is incremented) → MBR → IR

Fetch Phase in RTL Notation:



FETCH $[MAR] \leftarrow [PC]$

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

$[MBR] \leftarrow [[MAR]]$

$[IR] \leftarrow [MBR]$

Execute Phase (Register-to-Register Operation):

- If IR includes register numbers, the control unit will pass the values of the registers to the ALU (arithmetic logic unit), where arithmetic will be performed
- The result is passed to the destination register
- This is called a register-to-register operation because we take information from the registers, process it, and store it back in a register

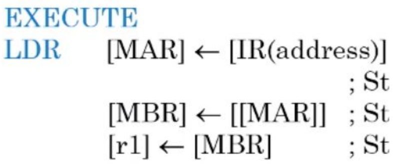
Registers (r0, r1) → ALU → Destination Register (ri)

Execute Phase (Memory Access):

- Eg. load or store
- If IR include memory address, this value is passed from IR to MAR where read/write is performed
- However, the MAR has 2 input data lines (one from operands, one from PC), so we need to use a Mux (multiplexer) to control flow, like a traffic light at an intersection
  - The control unit will select which input lines to take

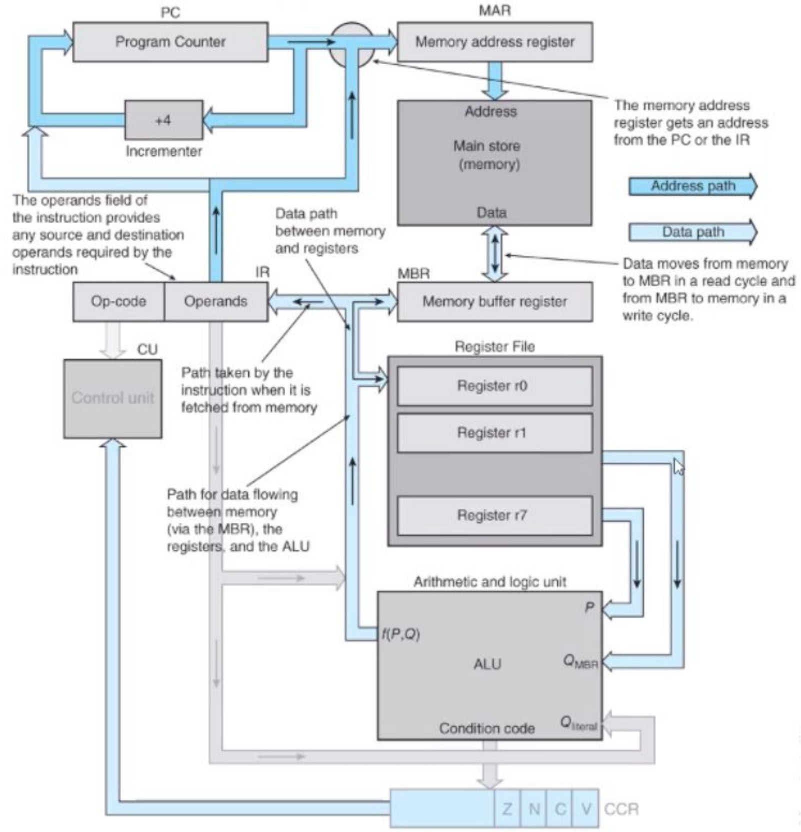
IR → MAR

Execute Phase in RTL Notation: (Load Example)



```
EXECUTE
LDR     [MAR] ← [IR(address)]
                              ; St
        [MBR] ← [[MAR]]      ; St
        [r1] ← [MBR]          ; St
```

FIGURE 3.2 Partial structure of a hypothetical stored program machine


PC
Program Counter
MAR
Memory address register
+4
Incrementer
Address
Main store
(memory)
Data
The memory address register gets an address from the PC or the IR
Address path
Data path
The operands field of the instruction provides any source and destination operands required by the instruction
Data path between memory and registers
Data moves from memory to MBR in a read cycle and from MBR to memory in a write cycle.
IR
Op-code
Operands
MBR
Memory buffer register
CU
Control unit
Path taken by the instruction when it is fetched from memory
Register File
Register r0
Register r1
Register r7
Path for data flowing between memory (via the MBR), the registers, and the ALU
Arithmetic and logic unit
P
f(P,Q)
ALU
Q_MBR
Condition code
Q_literal
Z
N
C
V
CCR

look at slides if you want a more visual representation of this image

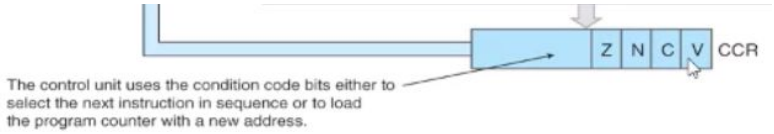
Constants

- Constant = a literal
- #25 → the # means that whatever comes after the # is going to be a constant/a literal
- The stored computer program sends constants to the ALU, the registers, or the MBR
    - Does not send to MAR because the constant is data, not an address

Flow Control

- Generally, a stored program computer executes instructions one after the other

- If we want to change the flow of execution, we need flow control which will change sequence of instructions
- Conditional Behaviour → allows a processor to either continue executing the next instruction or load the PC with a new value to execute another instruction that is not the preceding instruction
    - ie. we need to change the value of the PC



The control unit uses the condition code bits either to select the next instruction in sequence or to load the program counter with a new address.

- When the computer uses the ALU and performs an operation, it stores the status or condition information in the CCR
- This status information includes:
    - Z (zero)
    - N (negative in two's complement terms)
    - C (generated a carry-out)
    - V (generated an arithmetic overflow)
- If the result calculated by the ALU is any one of these status bits, it will set the bits to 1; otherwise, it will set them to 0 (like indicating whether the bits are active or not)

ARM (Advanced RISC Machines)

- 32-bit microprocessors
- 16 bits is a half-word, 32 bits is a word
- Register-to-register architecture
- Operand values are usually 32-bit wide
- Has 16 32-bit registers; r0, r1, r2, …, r12
    - R15 = PC (special purpose)
    - R14 = link register (holds subroutine return addresses) (special purpose)
    - R13 = reserved for use by programmer as stack pointer

Two Types of Computers

- CISC (Complex Instruction Set Computer)
    - All flags are automatically set after each operation
    - Typically have 2-address instructions where one address is memory, the other is a register
- RISC (Reduced Instruction Set Computer) like ARM processors
    - Programmer needs to decide to set flags after each operation or not
    - If the programmer wants to set the flags, they need to put an 'S' after the operation they input
        - Eg. SUB = subtraction, but SUBS = subtraction and set all flags
    - Typically have 3-address data processing instructions, where the three operands are registers
    - They have 2 special 2-address instructions: LOAD (LDR) and STORE (STR)

Sample ARM Assembly Instructions

1. LDR r0,address

Load the content of memory location at address into register r0.
Note: we have to have a space between the function (eg. LDR) and the first operand, but after that it is up to the programmer to put spaces or not.

2. STR r0,address

Store the contents of register r0 into the memory location at address.

3. ADD r0,r1,r2

Add the contents of registers r1 and r2 and store the result in r0.
Note: if you want to save flags, you have to put an ‘S’ after ADD

4. SUB r0,r1,r2

Subtract the contents of register r2 from r1 and store result in r0.
Note: if you want to save flags, you have to put an ‘S’ after SUB

5. BPL target i7 >,0

‘B’ means branch; ‘PL’ means plus
If the result of the previous operation was positive or zero, branch to the instruction at address target.

6. BEQ target i7 =

‘EQ’ means equal
If the result of the previous operation resulted in the Z flag = 1 (hence the result was 0), branch to the instruction at address target.

7. B target

No condition after B; means go-to, no matter any condition.
Go to the instruction stored at the address target.

8. MOV r0,#1

Copies the content of the second operand (in this case #1, but can be constant or register) into the first operand, the destination register r0.

- Needs only 2 operands

9. CMP r0,#21

CMP subtracts the second operand (in this case #21) from the register value, r0. It automatically updates the condition flags, but doesn’t place the result of the subtraction in a register; it ignores the result

- Needs only 2 operands

!= ≠.

10. BNE target

Branch not equal; if the value is not equal to something, jump to target.

11. MLA r0,r2,r1,r0 (multiply and accumulate)

Multiply r2 and r1 to each other, add it with r0, and store this value in r0.

Note: when you jump to another instruction, you continue executing the instructions in order following the instruction we jumped to.

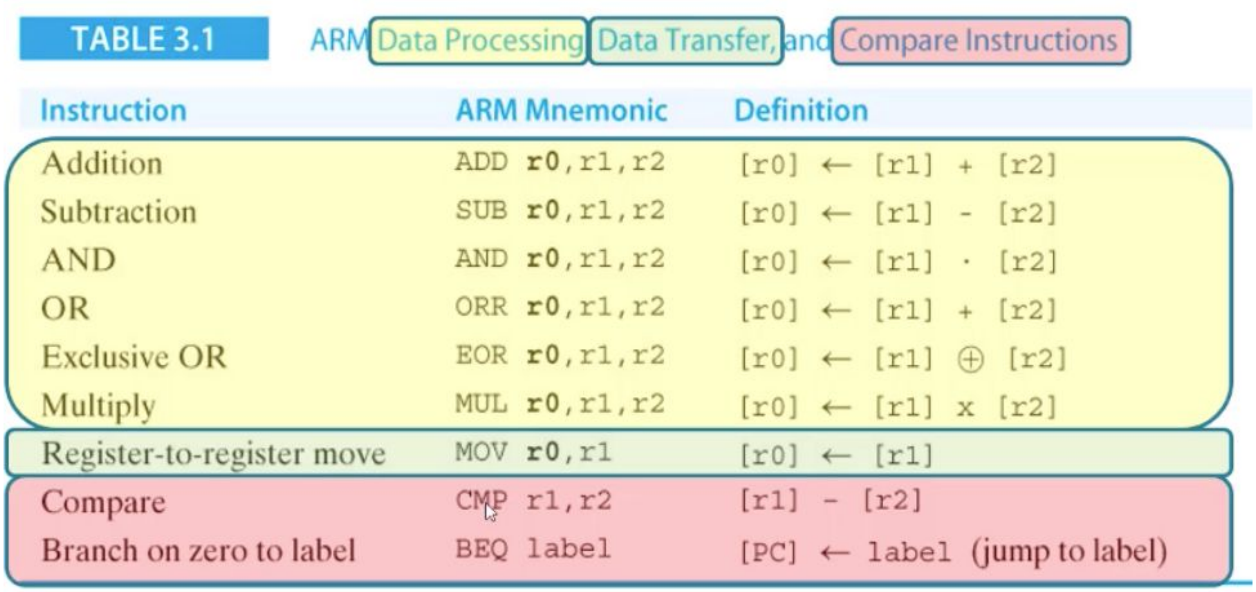
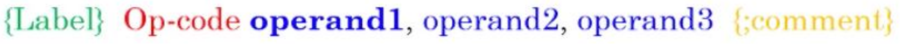


TABLE 3.1 ARM Data Processing, Data Transfer, and Compare Instructions

Instruction	ARM Mnemonic	Definition
Addition	ADD **r0**,r1,r2	[r0] ← [r1] + [r2]
Subtraction	SUB **r0**,r1,r2	[r0] ← [r1] - [r2]
AND	AND **r0**,r1,r2	[r0] ← [r1] · [r2]
OR	ORR **r0**,r1,r2	[r0] ← [r1] + [r2]
Exclusive OR	EOR **r0**,r1,r2	[r0] ← [r1] ⊕ [r2]
Multiply	MUL **r0**,r1,r2	[r0] ← [r1] x [r2]
Register-to-register move	MOV **r0**,r1	[r0] ← [r1]
Compare	CMP r1,r2	[r1] - [r2]
Branch on zero to label	BEQ label	[PC] ← label (jump to label)

How to Write ARM Assembly Language

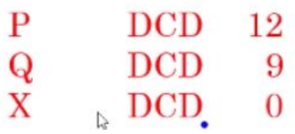


{Label} Op-code **operand1**, operand2, operand3 {;comment}

- If you don’t include a label, make sure to add a space before you write the Op-code so that the program recognizes it is an Op-code
- If an operation has less operands than required, it will repeat the first operand after it
- Label → user-defined, case-sensitive, single word without space
  - Can be used by other instructions to refer to the address of that line

Define Constant Data (DCD)

- DCD is an instruction to reserve memory for variables
- When we use DCD, we have to initialize the variables



```
P    DCD   12
Q    DCD    9
X    DCD    0
```

- Here, we initialize P with 12, Q with 9, and X with 0

Steps to Implement an Algorithm

1. Use DCD to reserve memory and initialize variables
2. Use LDR to load contents of variables into registers
3. Perform needed operations
4. If needed, use STR to store results into a given memory location
5. Last line of algorithm needs to be STOP

Note: you cannot perform operations on memory locations, only between registers and constants; this is why we load the variables into registers first

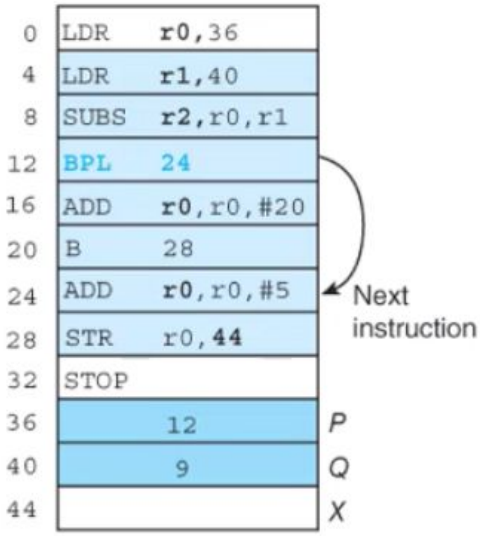
Note: for if/then statements where we may want to check the result of an operation, do not forget to write the ‘S’ at the end of the operations so the flags are stored

Comments in ARM Assembly Language

- Use a semicolon ; (done on the same line)
- After the semicolon, you can write comments in English or in RTL
- Make sure to write comments after each instruction

How to Calculate Memory Locations of Instructions/Variables

- Each instruction in ARM is 4 bytes
- We start at memory location 0; each line in the algorithm has its own memory location and increases by 4 as you go down the program (just like in CS2211)



```
 0  LDR   r0,36
 4  LDR   r1,40
 8  SUBS  r2,r0,r1
12  BPL   24
16  ADD   r0,r0,#20
20  B     28
24  ADD   r0,r0,#5
28  STR   r0,44
32  STOP
36        12        P
40        9         Q
44                  X
```

General-Purpose Registers

- You can put anything in these registers

Special-Purpose (Dedicated) Registers

- You can only put specific things in this register

ARM has general-purpose registers and 2 special purpose registers, usually hardware-defined registers (PC and link register).

Data Extension

Sometimes, registers hold data values that are smaller than their actual size (eg. holding a 16-bit value in a 32-bit register)

The way this is handled is processor dependent:

- Some leave the unused bits unchanged
- Some will fill the unused bits with 0s
- Some will extend the sign of the data value to the rest of the unused bits (ie. filling the unused bits with 1s or 0s, depending on sign)

Addressing Modes

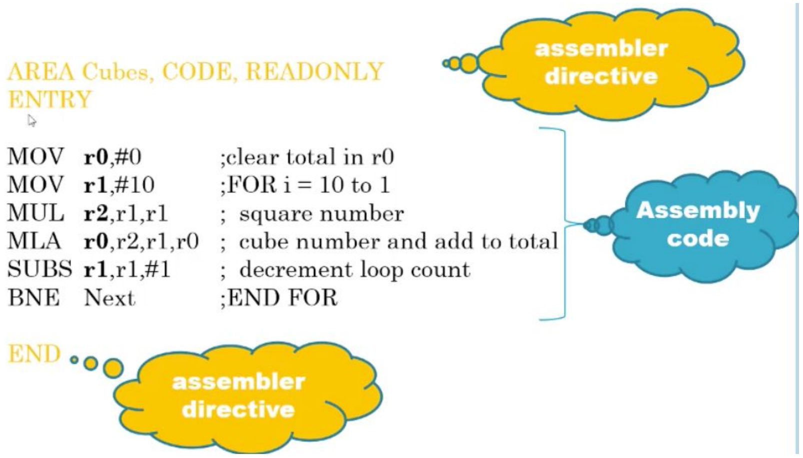
- Literal or immediate
    - Using the actual value (constant/literal) in the instruction
- Direct or absolute
    - Using the address of the data in the instruction
    - ARM doesn't support direct memory access
- Register indirect or pointer based or indexed
    - Using a register, which contains a memory address, to access memory
    - R1 is a pointer to the memory location

Instruction Types

- Memory to register
    - Source operands are in memory
    - Destination operand is in a register
    - Supported by CISC
- Register to memory
    - Source operands are in registers
    - Destination operand is in memory
    - Supported by CISC
- Register to register
    - Both operands are in registers
    - Supported by ARM

Assembly Directives

Syntax that describes the environment of the program; we need these with an algorithm in order to make it a functioning program.



```
AREA Cubes, CODE, READONLY
ENTRY

MOV  r0,#0        ;clear total in r0
MOV  r1,#10       ;FOR i = 10 to 1
MUL  r2,r1,r1     ;  square number
MLA  r0,r2,r1,r0  ;  cube number and add to total
SUBS r1,r1,#1     ;  decrement loop count
BNE  Next         ;END FOR

END
```

Assembly Program → made up of assembly directives and assembly instructions/code