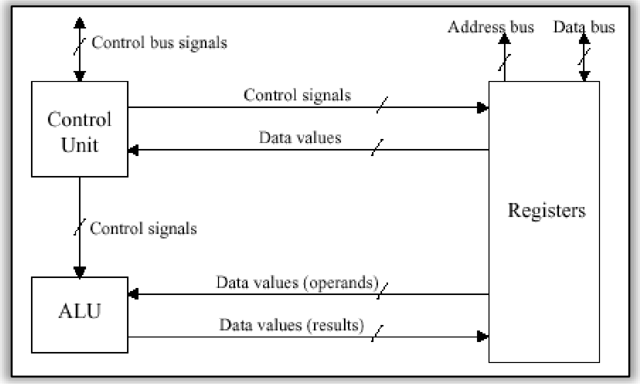
Xi'an Jiaotong-Liverpool University
西交利物浦大学

ARM Assembly – Part 1

Jianjun Chen (Jianjun.Chen@xjtlu.edu.cn)

Contents

- Introduction to ARM
- The VisUAL ARM emulator
- (Some) ARM Instructions



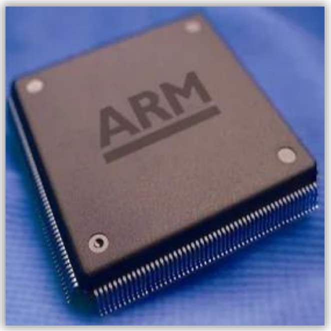
Introduction to ARM

Programmer’s view on the ARM architecture

Registers.

The CPU

- The central processing unit or CPU is the part of a microprocessor system that does all the ‘work’.
  - It interprets the instructions stored in memory.
  - It performs the calculations.
  - It controls the flow of data along the data bus.
  - It determines which memory address to use.



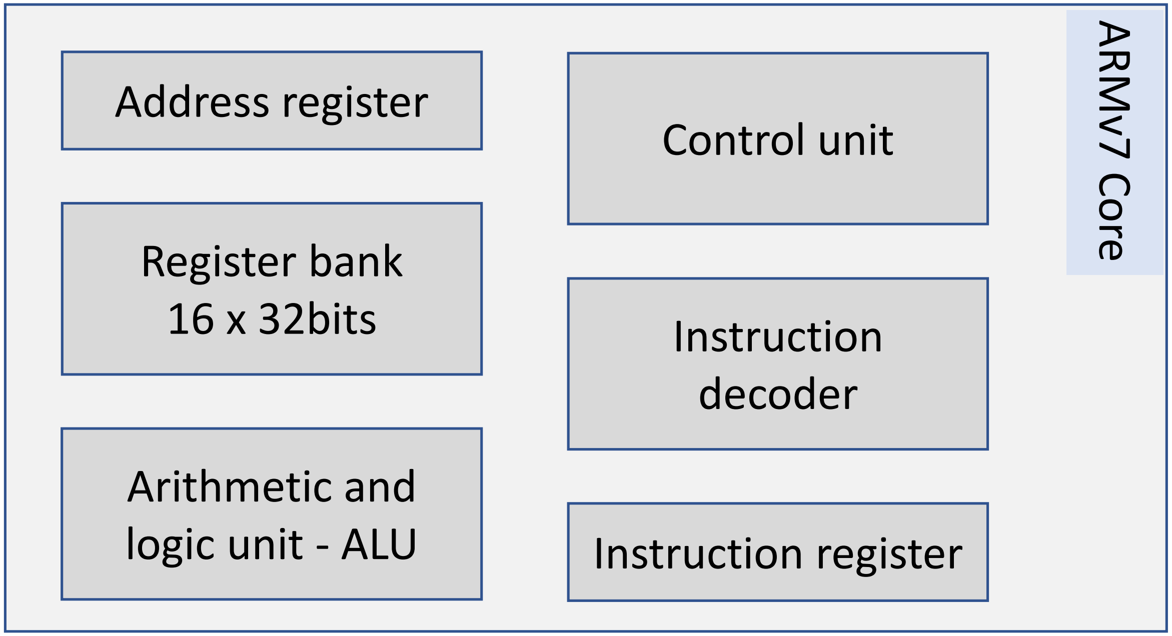
- The CPU is designed to perform all of these functions as efficiently as possible.

- It can be subdivided into a number of blocks, each with a distinct function.

- Every CPU is different and we will concentrate on the ARMv7 ‘core’ - the CPU for the ARMv7 range of microprocessors.

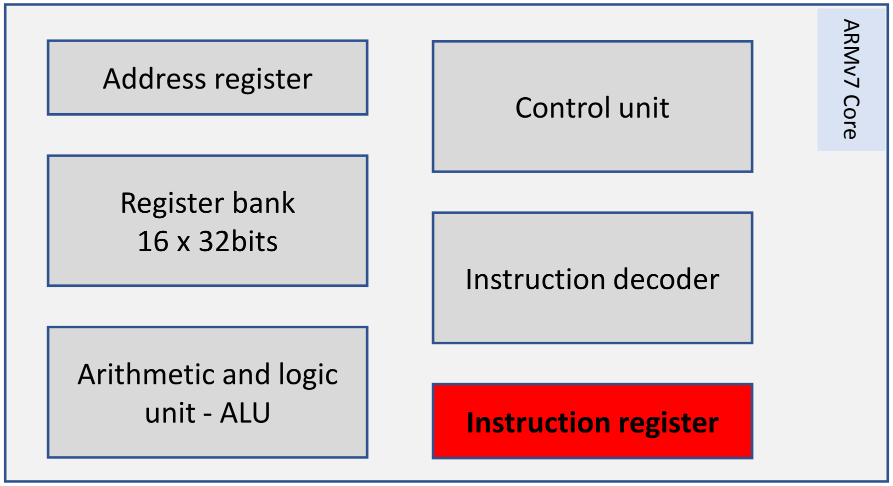
The ARMv7 Core

- The basic building blocks of the ARMv7 core are:



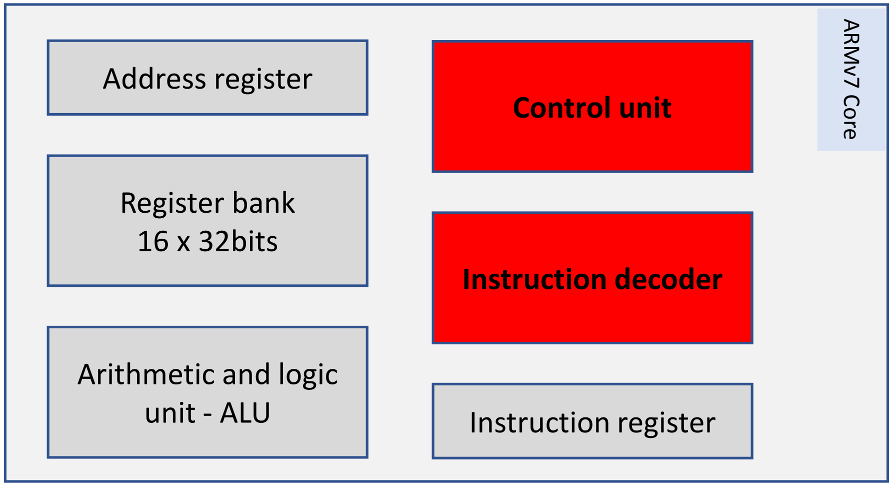
The Instruction Register

- The instructions stored in memory travel along the data bus to the CPU where they are loaded into the instruction register.
- ARM7 instructions are 32 bits long so the instruction register is a 32 bit memory device - not part of the main memory.
- The process of loading the instruction register from memory is known as a ‘**fetch**’.



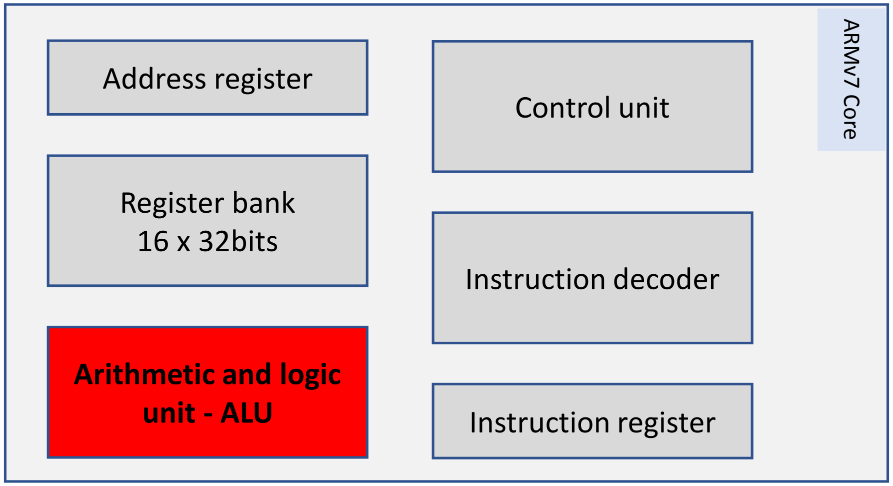
Instruction Decoder, Control Unit

- The instructions are in ‘machine code’ and the instruction decoder determines the function of each instruction.
- The instruction decoder and control unit determine what the other parts of the CPU do.
- The control unit is also in charge of the control bus.
- The process of interpreting each instruction is known as the ‘**decode**’ cycle.



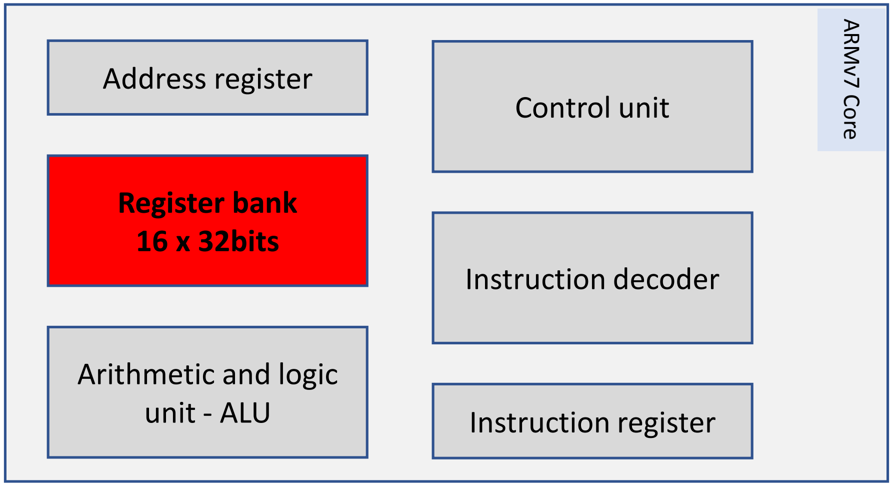
Arithmetic and Logic Unit

- The arithmetic and logic unit or ALU performs the mathematical functions as required.
- These may be arithmetic such as add, subtract or multiply or logical such as AND, OR, XOR etc.
- The process of performing each instruction is known as the ‘**execute**’ cycle.



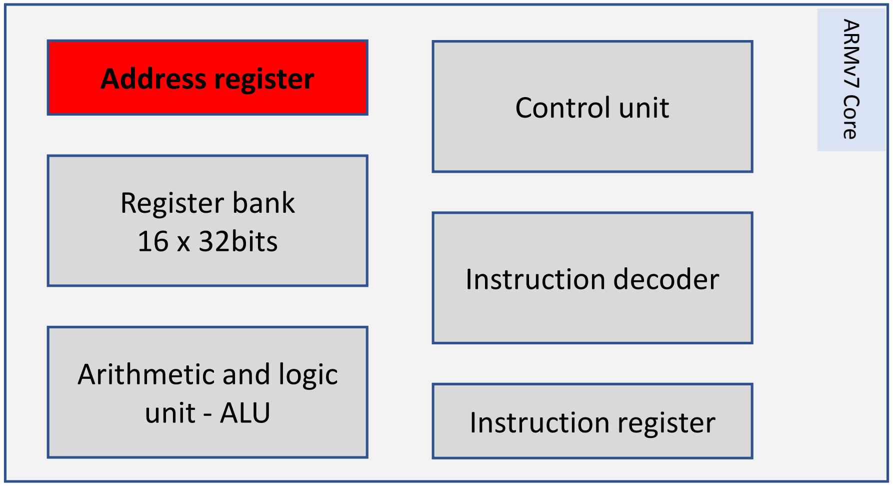
Register Bank

- The register bank is a local memory for the CPU. It has 16 locations - each location can hold 32 bits of data.
- The registers are named r0, r1, r2, r3, ... etc. up to r15.
- They are used to hold data which is processed by the ALU and also hold the results of any calculation.
- Registers r13, r14 and r15 have special functions which we will cover later



Address Register

- The address register is a 32 bit memory device which holds a memory address value.
- Either this address may be for the memory location of the next instruction during the ‘fetch’ cycle.
- Or during the ‘execute’ cycle the address is for a memory location either containing data to be loaded into a register or where data from a register is to be stored.

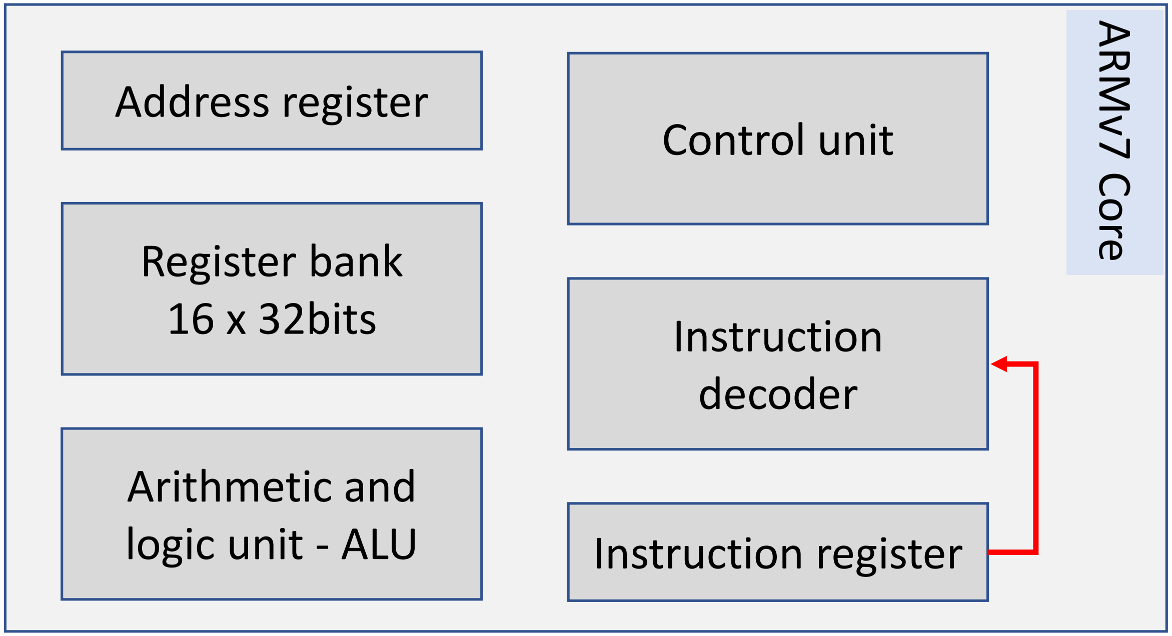


Fetch, Decode, Execute.

- The CPU performs three cycles sequentially
- During the **fetch cycle** an instruction in memory is loaded into the instruction register.
- During the **decode cycle** the instruction is interpreted by the instruction decoder.
- During the **execute cycle**
  - Either the ALU performs a calculation on values held in registers or
  - A value in a register is stored into memory or
  - a value in memory is loaded into a register

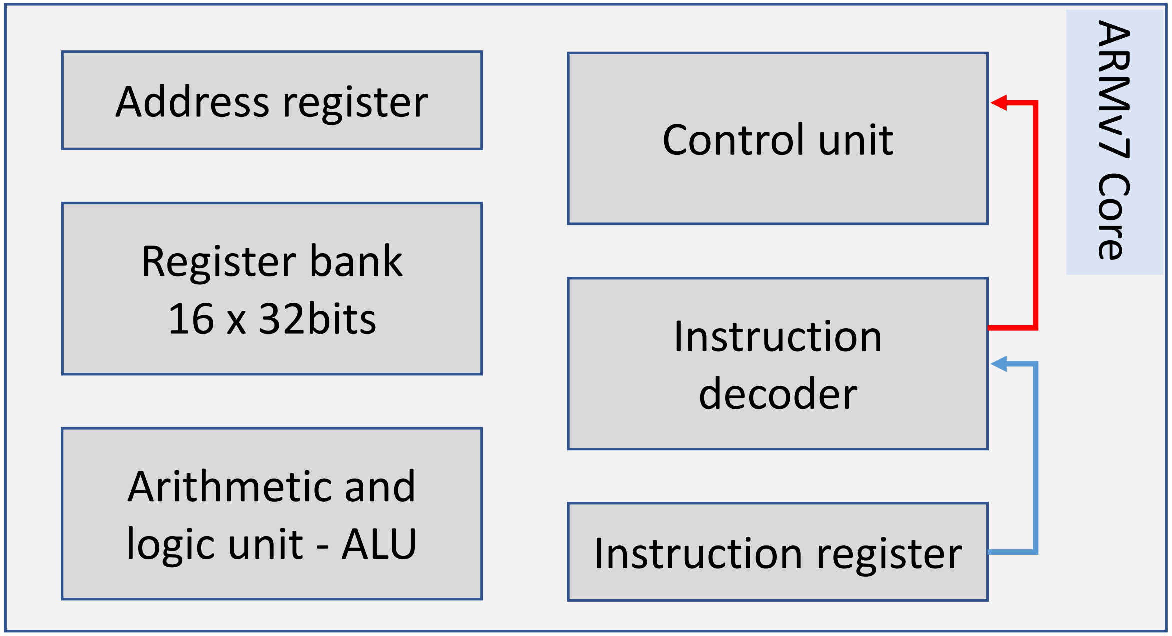
Internal Connections

- The instruction register is connected to the instruction decoder.



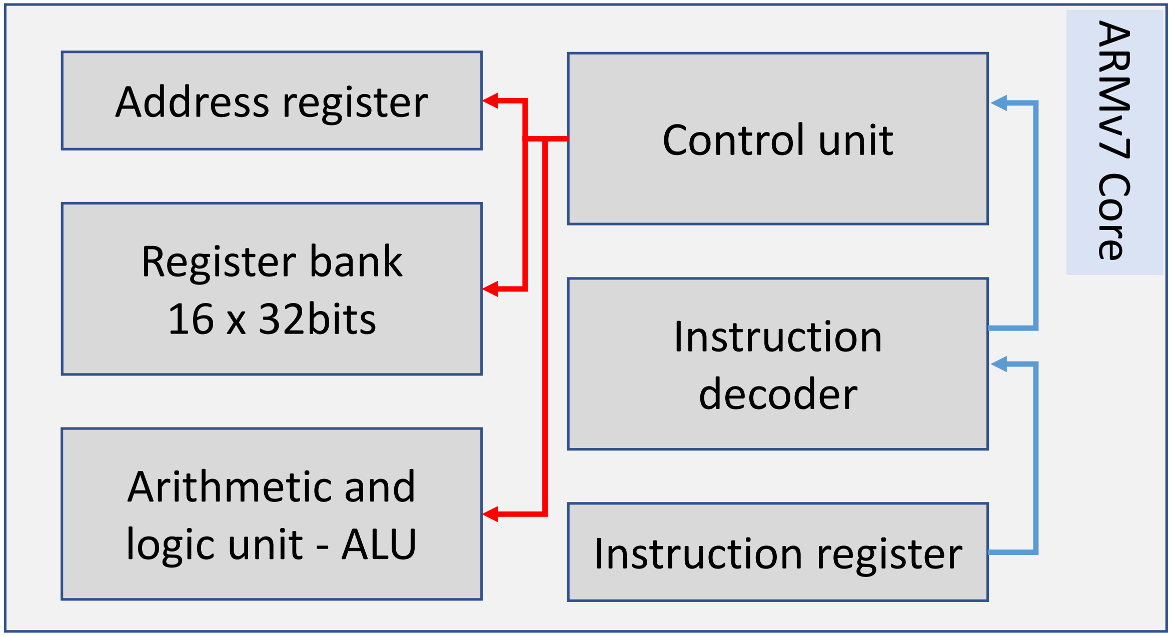
Internal Connections

- The instruction decoder is connected to the control unit.



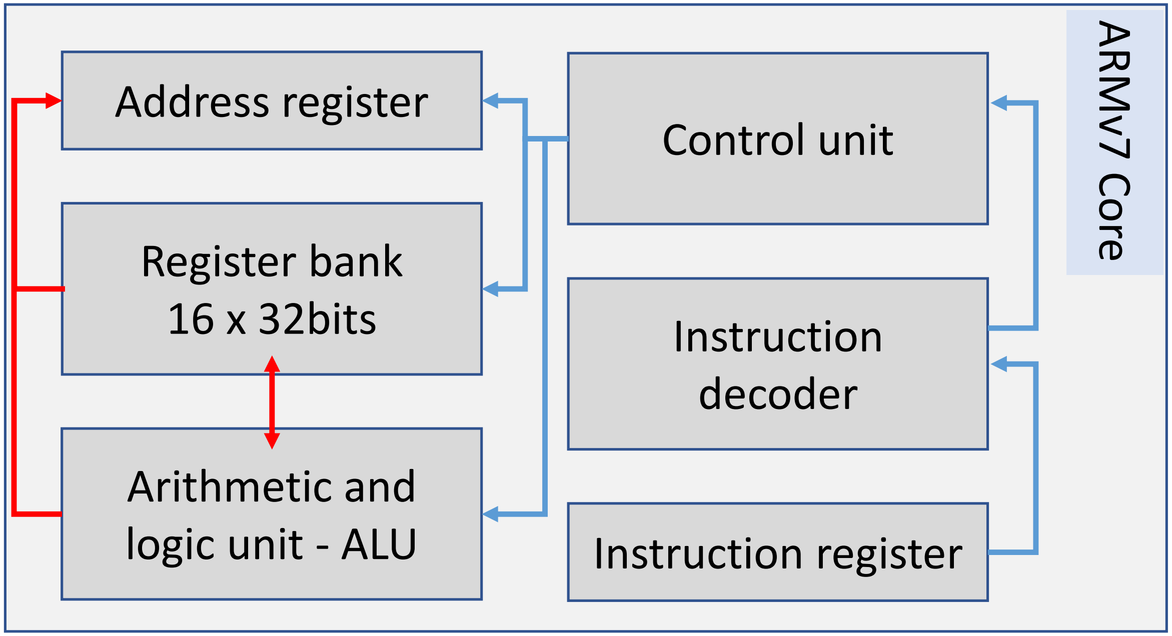
Internal Connections

- The control unit is connected to the ALU, the register bank and the address register



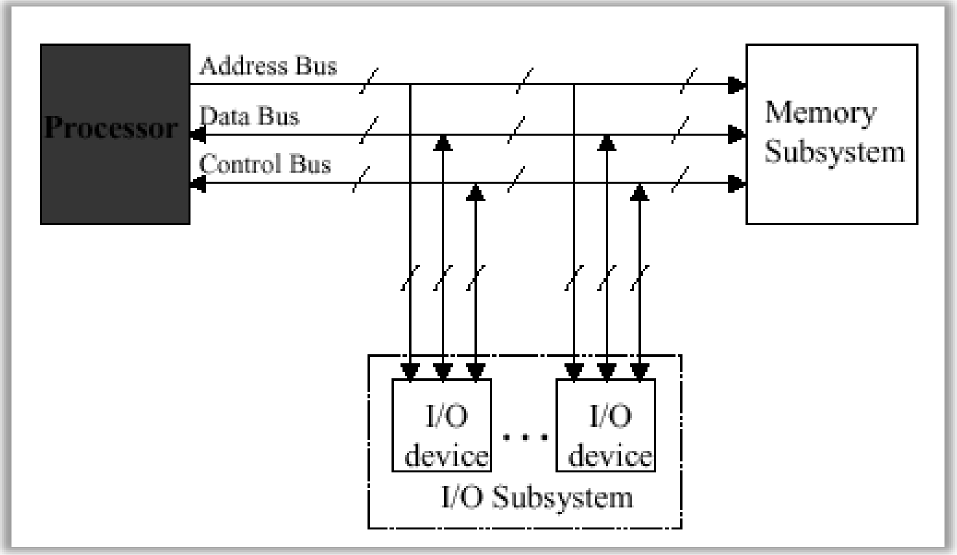
Internal Connections

- The ALU and the register bank are connected to each other and the address register.



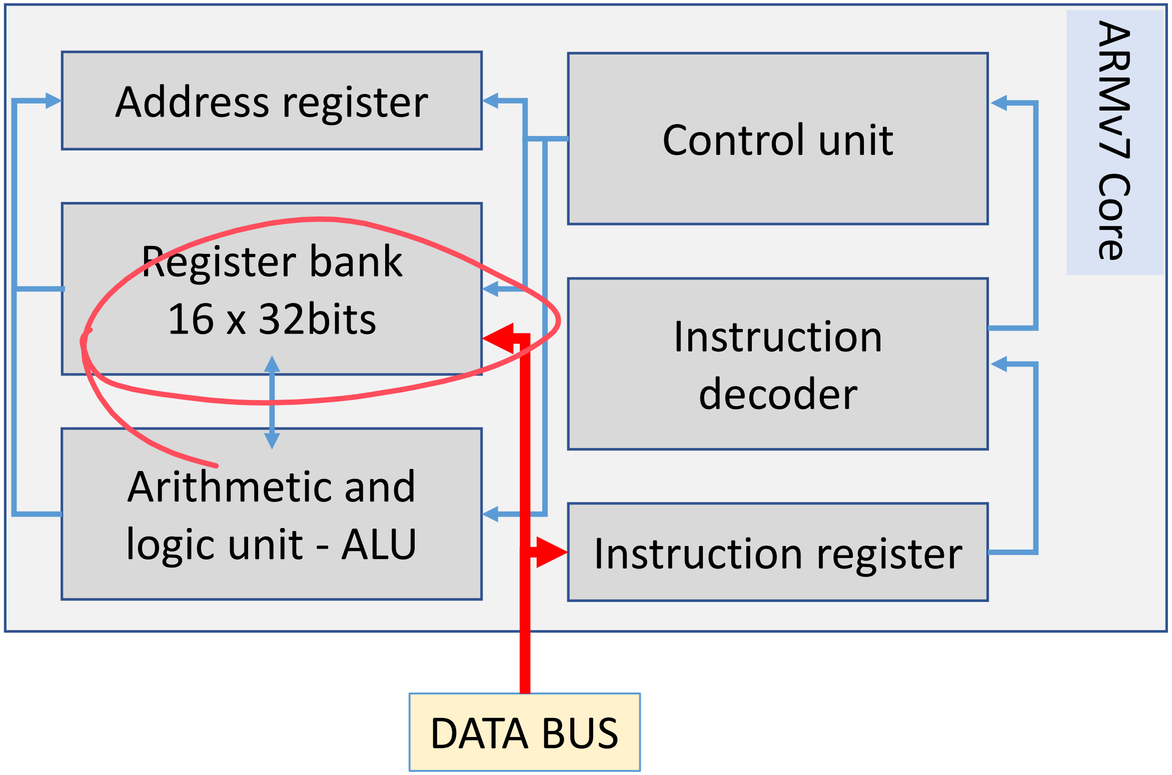
Data Bus

- A bus is a collection of electrical connections:
  - Normally 8, 16, 32 or 64 individual wires.
  - 32 bits of data can pass along a 32-bit bus at the same time.
- A **data bus** can be connected with many different devices.
  - Devices have only one connection onto the data bus.
- Devices pass/receive data via the data bus.



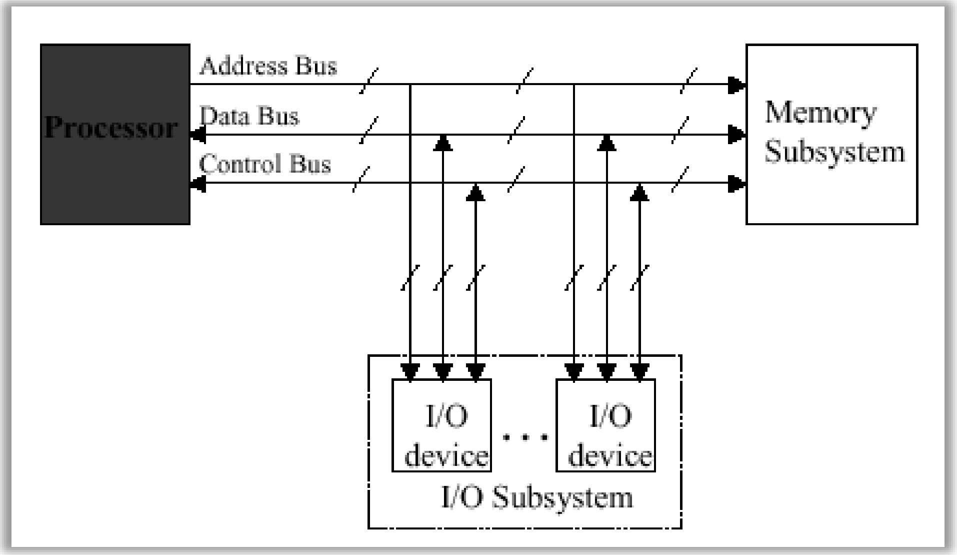
External Connections: Data Bus

- The data bus is connected to both the instruction register and the register bank.



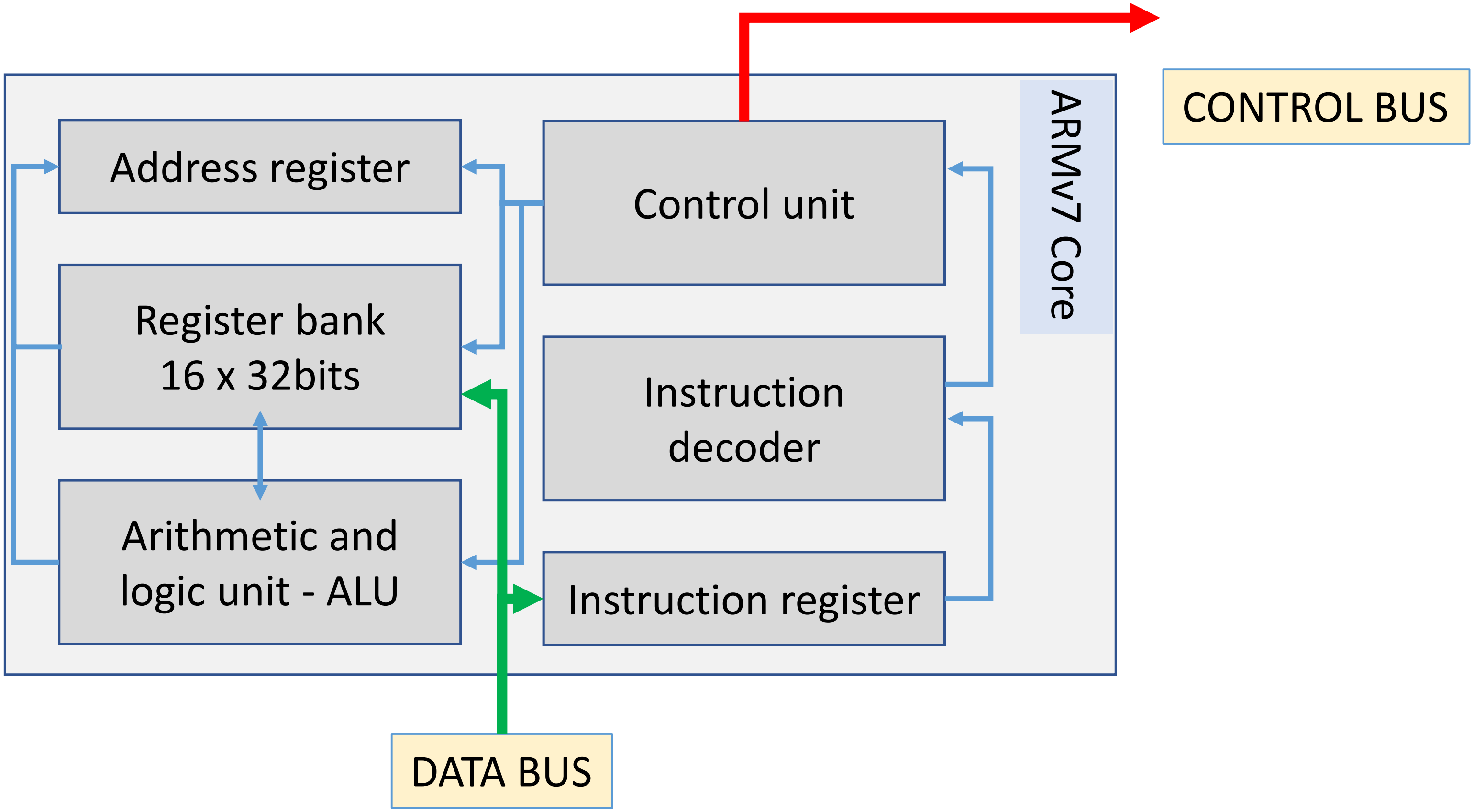
Control Bus

- It is important that signals do not collide on the data bus.
  - Only one device can send data at the same time.
- The CPU controls all movements on the bus using special wires to activate devices and to synchronize the sending and receiving devices.
- These special wires are called **control bus**.



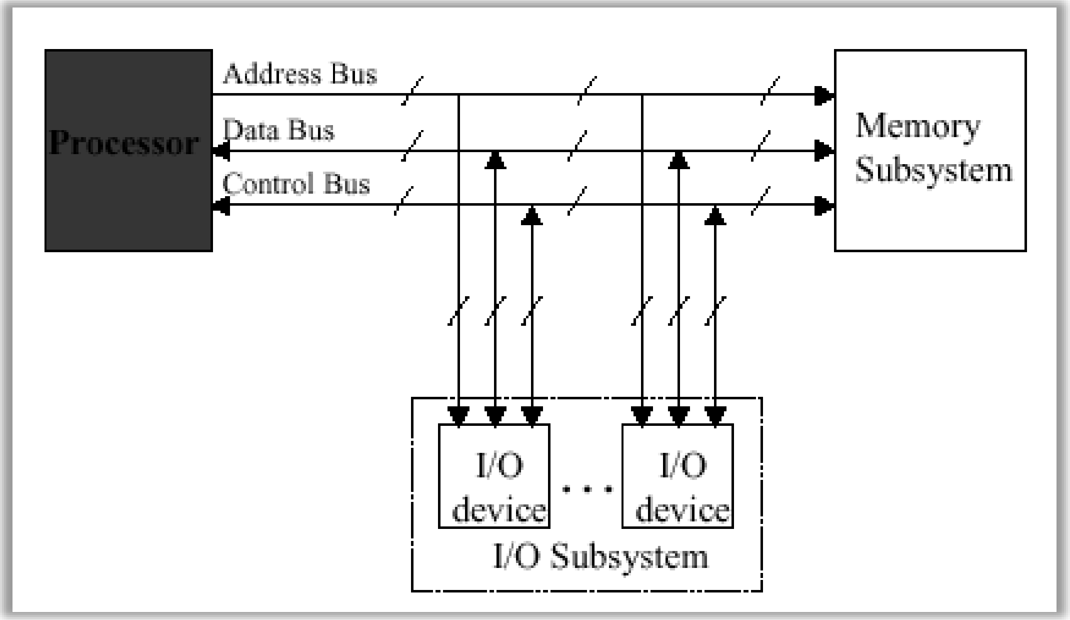
External connections

- The control bus is connected to the control unit.



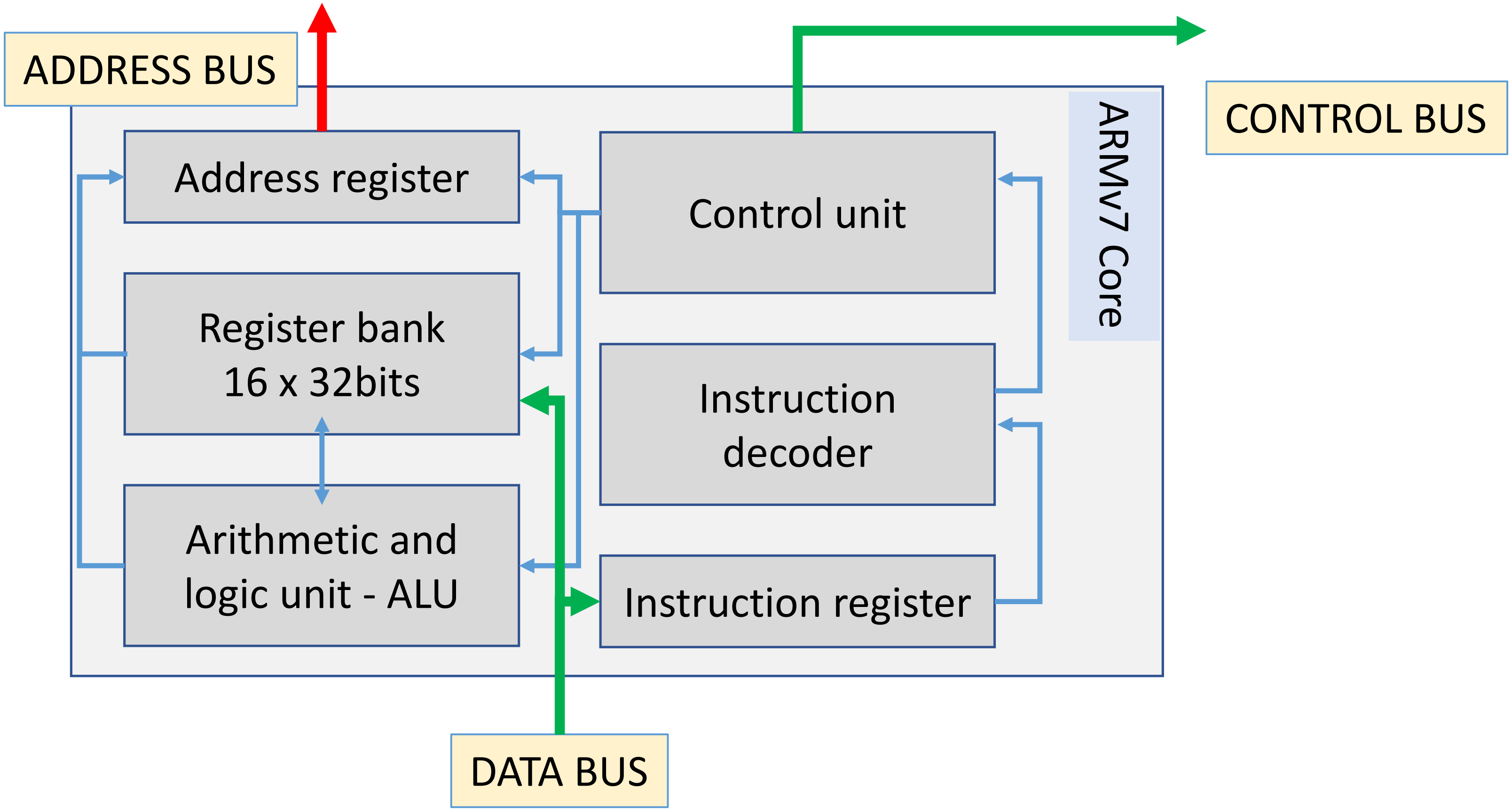
Address Bus

- In addition to the data bus and control bus, there is a third bus called the address bus.
- The address bus is used by the CPU to determine which location in memory is sending or receiving data.



External connections

- The address bus is connected to the address register



ARM Registers

General purpose registers: R0 ~ R12

Special registers: PC, SP, LR, CPSR

ARM Registers

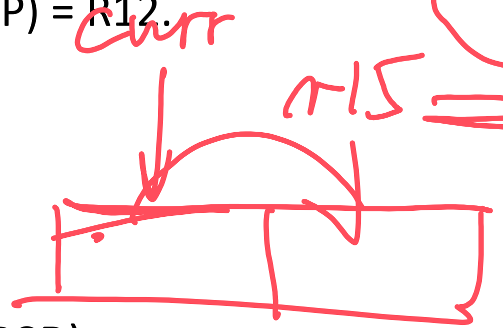
- ARM processors provide general-purpose and special-purpose registers.
  - Some additional registers are available in privileged execution modes. (Covered later)

- In all ARM processors, the following registers are available and accessible:
  - 13 general-purpose registers R0-R12.
    - Intra-Procedure-call scratch register (IP) = R12.
    - Frame Pointer (FP) = R11.
  - 1 Stack Pointer (SP) = R13.
  - 1 Link Register (LR) = R14.
  - 1 Program Counter (PC) = R15.
  - 1 Current Program Status Register (CPSR).

GP

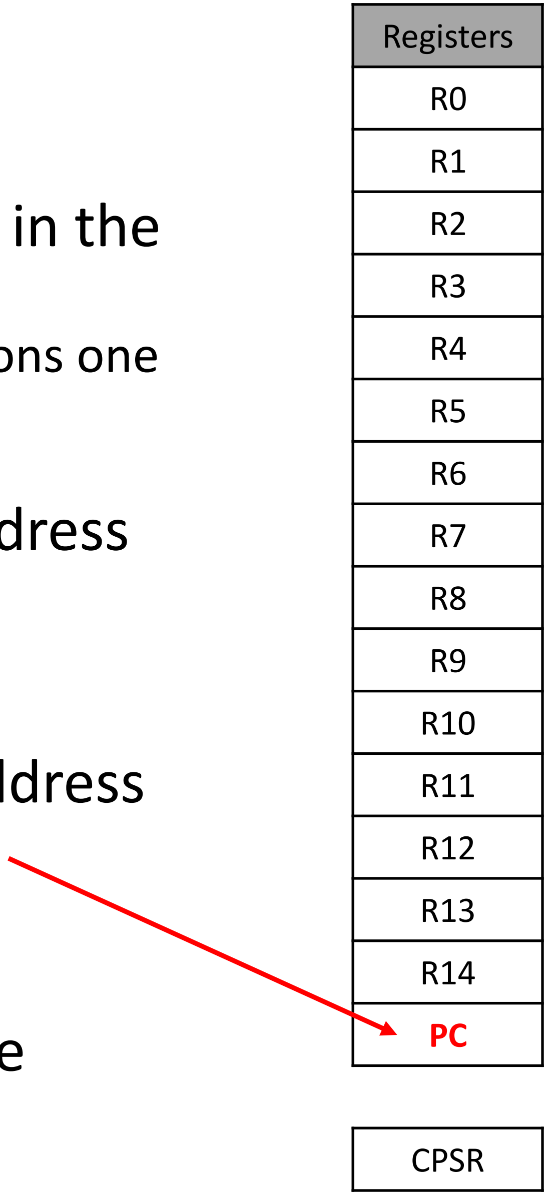
Registers
R0
R1
R2
R3
R4
R5
R6
R7
R8
R9
R10
R11
R12
R13
R14
PC

CPSR



Program Counter

- Instructions are “commands” that stored in the main memory.
  - CPU fetch, decode and then execute instructions one by one.
- How does the CPU know the memory address for the next instruction in the computer program?
- Register r15 always holds the memory address of the next instruction to be executed.
  - Written in many books, but not exactly so.
- An alternative name for register r15 is the ‘program counter’.



Registers

R0

R1

R2

R3

R4

R5

R6

R7

R8

R9

R10

R11

R12

R13

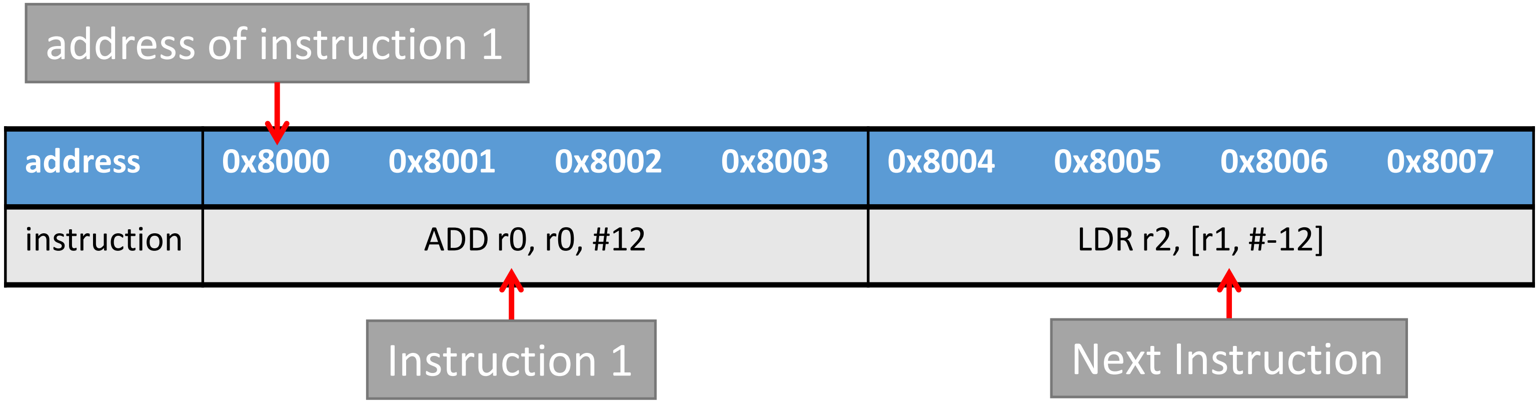
R14

PC

CPSR

Instructions Stored in Memory

- Instructions are usually 32 bits long, whereas each memory address value refers to 1 byte (8 bits long).
- Thus, the address of one instruction points to 4 bytes starting from that address.
- In general instructions are in consecutive locations in memory so that they are executed in the same order as they appear in memory.



address of instruction 1

address	0x8000	0x8001	0x8002	0x8003	0x8004	0x8005	0x8006	0x8007
instruction	ADD r0, r0, #12				LDR r2, [r1, #-12]			

Instruction 1

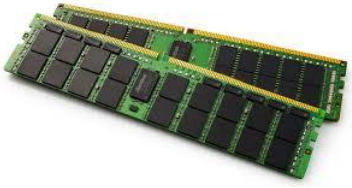
Next Instruction

Program Counter and Instructions

- When an instruction is executing, the program counter, r15, increments by 4 so that it holds the memory address of the next instruction.

- EXCEPT when a ‘branch’ instruction is executed when the computer program ‘branches’ to another part of memory and the program counter holds a completely new memory address.
  - A branch instruction is similar to ‘goto/if else’ in C.

Instructions



- ARM belongs to “reduced instruction set computing” (RISC) architecture.
  - Use simple instructions that can be executed within one clock cycle.
    - A 3 GHz processor performs 3,000,000,000 clock cycles per second
  - Instruction format is simple and each instruction does a simple task.
  - More transistors for registers -> more registers

- The other type of architecture is called “complex instruction set computing” (CISC) architecture.
  - One instruction can do multiple tasks, within multiple clock cycle.
    - Less memory usage for instructions.
    - More transistors needed to implement all instructions.
  - Less transistors for registers -> less registers.

Example difference

- RISC CPU only processes data in registers, while CISC can directly process data in Memory.
  - E.g., for RISC CPUs, data must be first loaded from the memory. After processing data, CPU needs an additional instruction to store the data to the memory.

C/Java Code	x86 assembly	arm7tdmi
x = x + 5;	addl $5, -4(%rbp)	ldr r3, [fp, #-8] add r3, r3, #5 str r3, [fp, #-8]

load

add

save

- Issue of using more memory:
  - Can be solved by providing more memory, but infeasible for some embedded systems
  - Can use thumb instructions when possible
- Issue of having more steps: Pipelining
  - Taught later this semester.

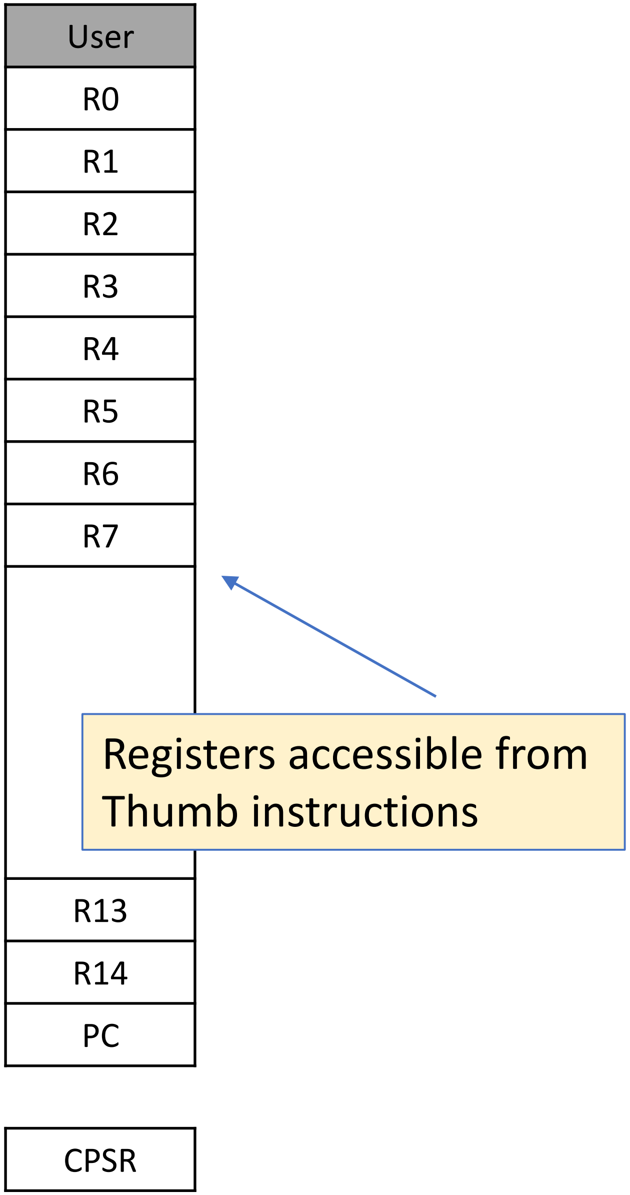
Thumb Instructions

- The Thumb instruction set is a subset of the most commonly used 32-bit ARM instructions.
    - ARM7TDMI: The “T” in the core's full name specifies Thumb.

- Thumb instructions are 16 bits long, and have a corresponding 32-bit ARM instruction that has the same effect on processor model.

- Thumb instructions can access less registers, but they also occupy less memory space.

- Some ARM CPUs for embedded boards only supports thumb instructions.



User

R0

R1

R2

R3

R4

R5

R6

R7

R13

R14

PC

CPSR

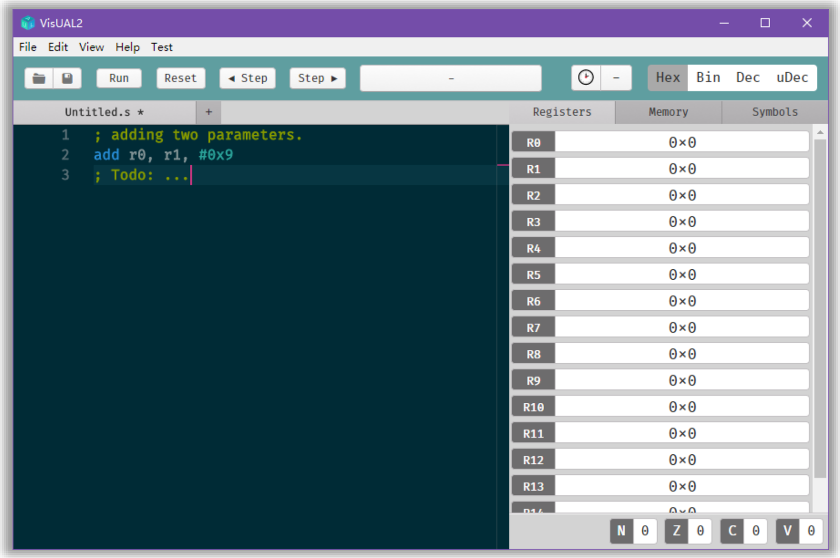
Registers accessible from Thumb instructions

VisUAL ARM Emulator

For labs and coursework

The VisUAL emulator

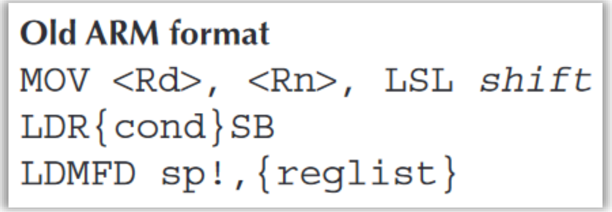
- For the labs and coursework, we will use a software called VisUAL.
    - Available in the shared folder on box
- UAL stands for "Unified Assembler Language"
    - is a common syntax for ARM and Thumb instructions.
- Code written using UAL can be assembled for ARM or Thumb for any ARM processor.



```
; adding two parameters.
add r0, r1, #0x9
; Todo: ...
```

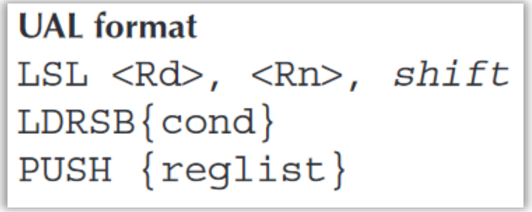
R0	0x0
R1	0x0
R2	0x0
R3	0x0
R4	0x0
R5	0x0
R6	0x0
R7	0x0
R8	0x0
R9	0x0
R10	0x0
R11	0x0
R12	0x0
R13	0x0

N	0	Z	0	C	0	V	0



Old ARM format

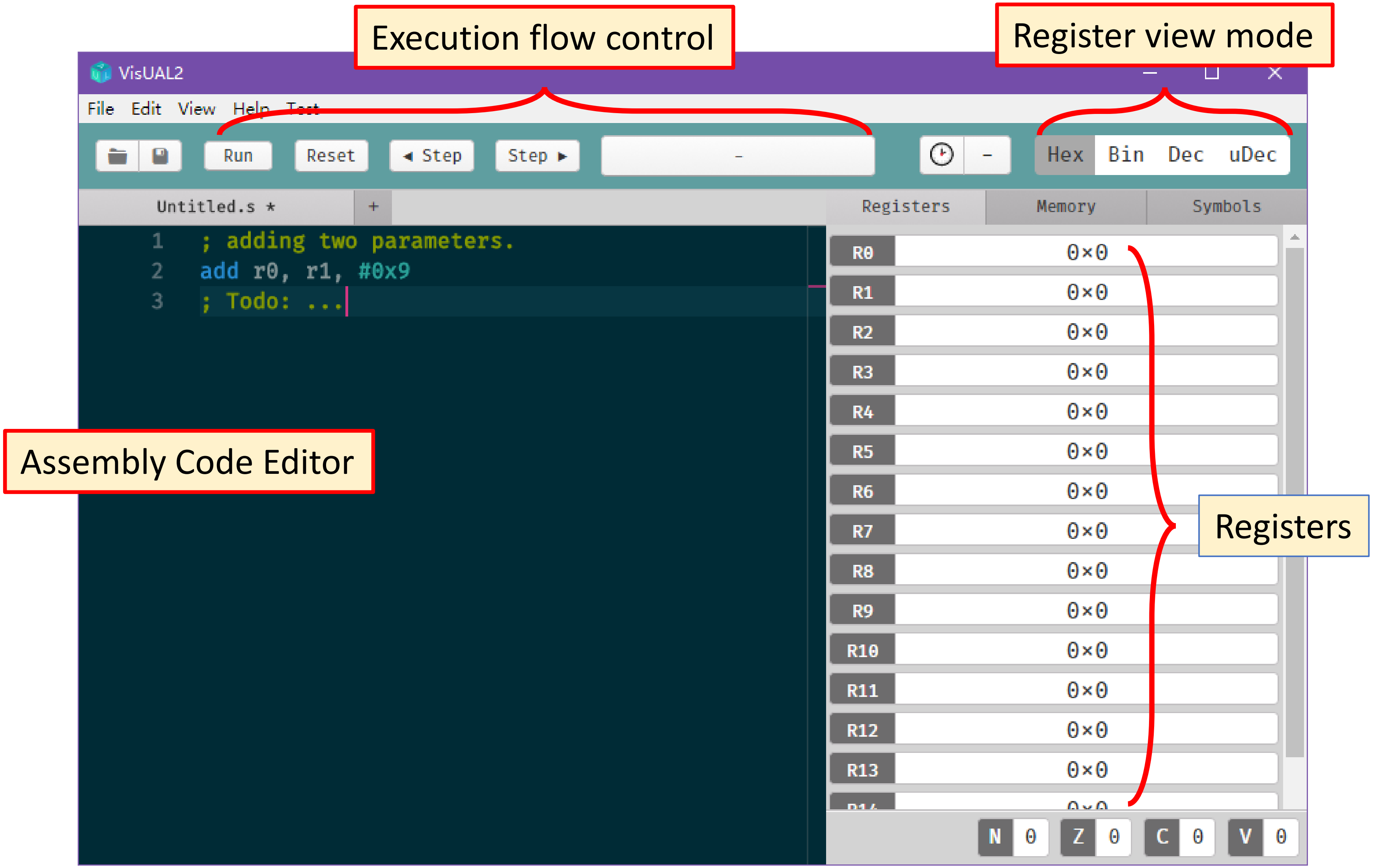
```
MOV <Rd>, <Rn>, LSL shift
LDR{cond}SB
LDMFD sp!,{reglist}
```



UAL format

```
LSL <Rd>, <Rn>, shift
LDRSB{cond}
PUSH {reglist}
```

The VisUAL emulator



Execution flow control

Register view mode

Assembly Code Editor

Registers

```
; adding two parameters.
add r0, r1, #0x9
; Todo: ...
```

R0	0×0
R1	0×0
R2	0×0
R3	0×0
R4	0×0
R5	0×0
R6	0×0
R7	0×0
R8	0×0
R9	0×0
R10	0×0
R11	0×0
R12	0×0
R13	0×0

N	Z	C	V
0	0	0	0

Supported Instructions and Registers

- VisUAL does not support all ARMv7 instructions.
  - But it is enough for the current teaching purpose.

- You can use gcc arm cross-compiler to generate real assembly code from C/C++ code.
  - Please explore this by yourself, I can help if you ask me.

- Check here for official documentation of VisUAL:
  - https://tomcl.github.io/visual2.github.io/

ARM Assembly: Basic Operations

Arithmetic, bitwise, shift.

immediate values,

Introduction to instruction encoding

Instruction Set Overview

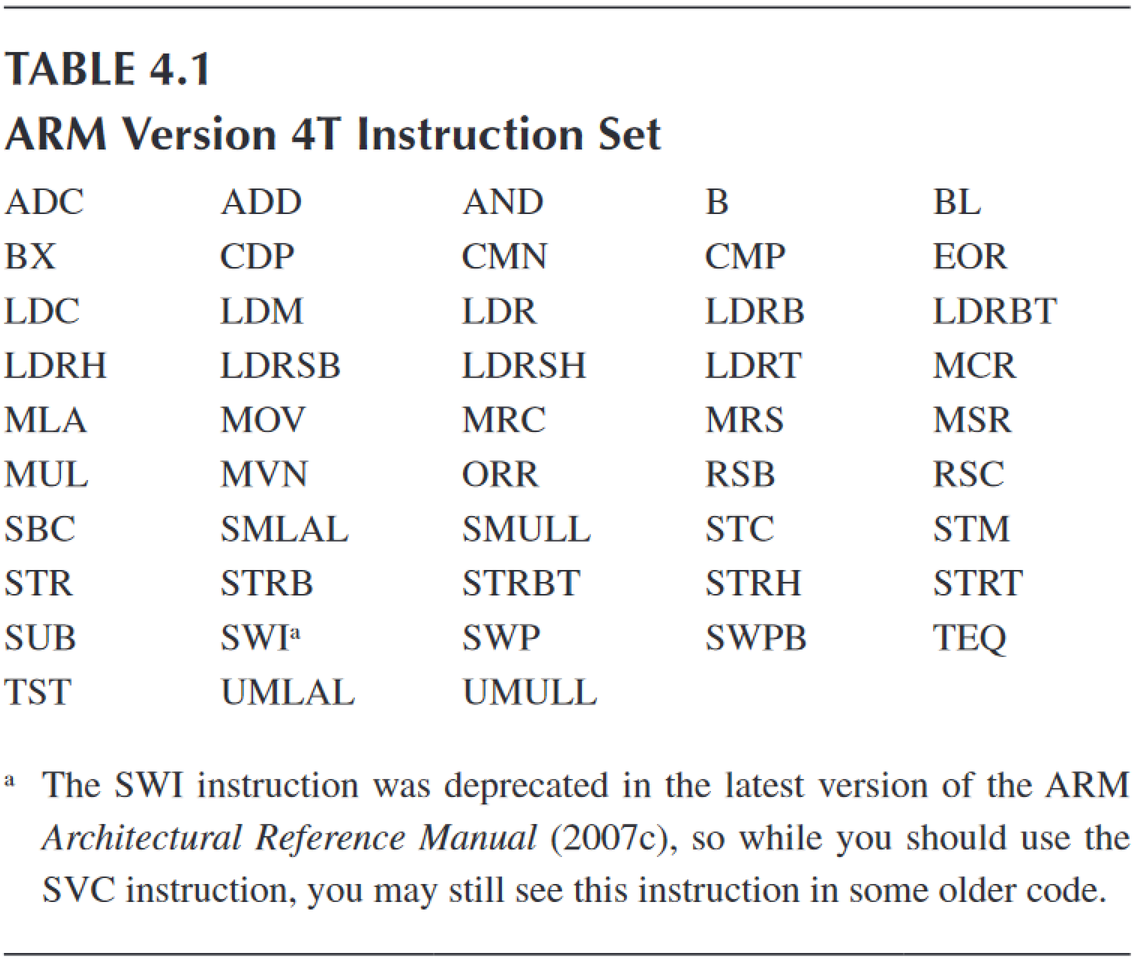
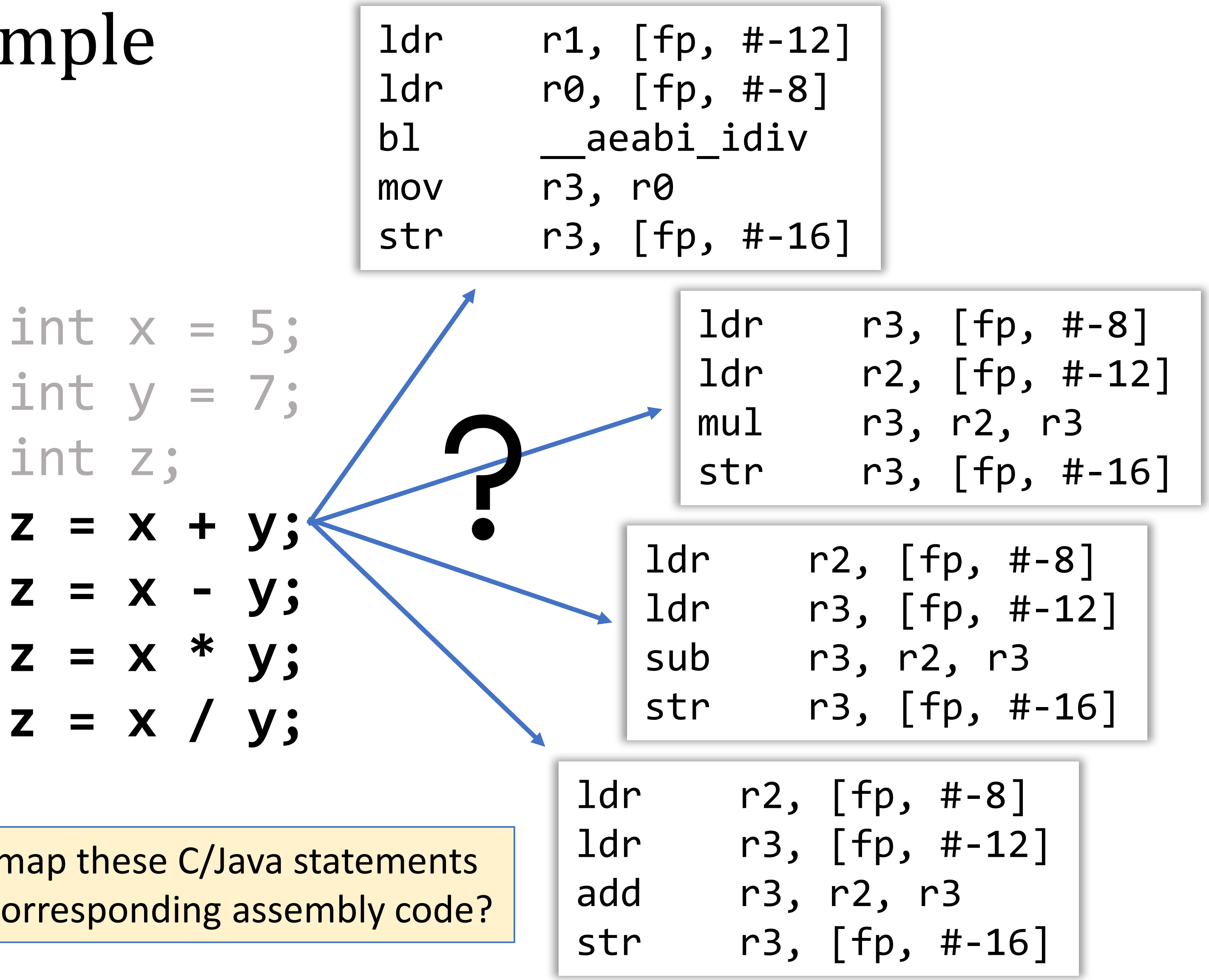


TABLE 4.1
ARM Version 4T Instruction Set

ADC	ADD	AND	B	BL
BX	CDP	CMN	CMP	EOR
LDC	LDM	LDR	LDRB	LDRBT
LDRH	LDRSB	LDRSH	LDRT	MCR
MLA	MOV	MRC	MRS	MSR
MUL	MVN	ORR	RSB	RSC
SBC	SMLAL	SMULL	STC	STM
STR	STRB	STRBT	STRH	STRT
SUB	SWI[a]	SWP	SWPB	TEQ
TST	UMLAL	UMULL		

[a] The SWI instruction was deprecated in the latest version of the ARM *Architectural Reference Manual* (2007c), so while you should use the SVC instruction, you may still see this instruction in some older code.

Example



```
int x = 5;
int y = 7;
int z;
z = x + y;
z = x - y;
z = x * y;
z = x / y;
```

?

```
ldr     r1, [fp, #-12]
ldr     r0, [fp, #-8]
bl      __aeabi_idiv
mov     r3, r0
str     r3, [fp, #-16]
```

```
ldr     r3, [fp, #-8]
ldr     r2, [fp, #-12]
mul     r3, r2, r3
str     r3, [fp, #-16]
```

```
ldr     r2, [fp, #-8]
ldr     r3, [fp, #-12]
sub     r3, r2, r3
str     r3, [fp, #-16]
```

```
ldr     r2, [fp, #-8]
ldr     r3, [fp, #-12]
add     r3, r2, r3
str     r3, [fp, #-16]
```

Can you map these C/Java statements to their corresponding assembly code?

Arithmetic Instructions

- The instructions below are for integer arithmetic:
  - Rd is the register where the result is saved
  - Rn is the register holding the first operand.
  - Op2 is the second operand, it can be a register or an immediate value (a number directly encoded inside instructions)

items between {} are optional

Name	Syntax
Add	`ADD{S}{cond} Rd, Rn, op2`
Subtract	`SUB{S}{cond} Rd, Rn, op2`
Reverse Subtract	`RSB{S}{cond} Rd, Rn, op2`
Multiply (Unsupported in VisUAL)	`MUL{S}{cond} {Rd}, Rn, Rm`
Division (Unsupported in ARMv7)	----------------------------

- Floating point arithmetic instructions will not be covered as this is usually done on floating point units (FPU).

Examples

- The (#) symbol indicates that the operand 2 is an immediate value.
- Only operand 2 can be immediate values.

Expression	Math meaning
add r0, r0, #12	r0 = r0 + 12
sub r2, r1, r0	r2 = r1 – r0
sub r2, r0, r1	r2 = r0 – r1
rsb r2, r1, r0	r2 = r0 – r1
sub r2, r0, #11	r2 = r0 - 11
rsb r2, r0, #11	r2 = 11 – r0

- For convenience, you can use dec, oct, bin, hex or even char as immediate: `ADD r1, r0, #'a'`
- Q: Why Reverse Subtract (RSB) instruction is needed?

The MOV Instruction

- The `mov` instruction copies a value from Operand2 to a register:
  - Operand2 can either be a register or an immediate.

```
MOV{S}{cond} Rd, Operand2
```

- For example:
  - `MOV r0, r1`
  - `MOV r0, #15`
  - `MOV r0, #'a'`
  - `MOV r0, #0x7FF`

Shift and Rotation Instructions

- Following shift-like instructions are available:
  - `Rd` is where the result is stored.
  - `Rm` is the register holding the first operand.
  - `Rs` is the register storing the second operand.
  - `#sh` is an immediate value between 0~31.

Name	Syntax

Logical Shift Left	`LSL{S}{cond} Rd, Rm, Rs`
	`LSL{S}{cond} Rd, Rm, #sh`

Logical Shift Right	`LSR{S}{cond} Rd, Rm, Rs`
	`LSR{S}{cond} Rd, Rm, #sh`

Arithmetic Shift Right	`ASR{S}{cond} Rd, Rm, Rs`
	`ASR{S}{cond} Rd, Rm, #sh`

Rotate right	`ROR{S}{cond} Rd, Rm, Rs`
	`ROR{S}{cond} Rd, Rm, #sh`

More Bitwise Instructions

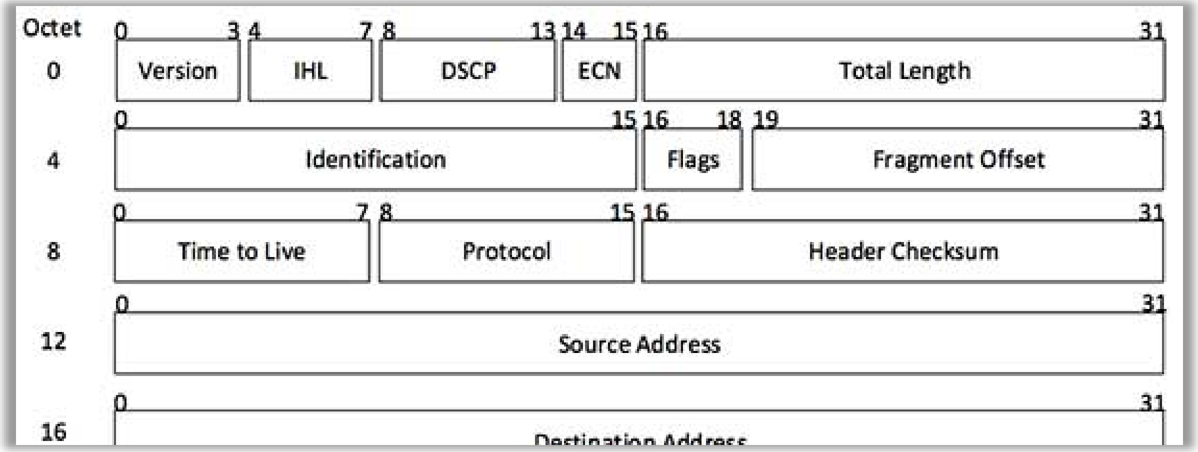
- Following bitwise instructions are available:
  - `Op2` can be a register or immediate.

Name	Syntax
Bitwise and	`AND{S}{cond} Rd, Rn, op2`
Bitwise exclusive or	`EOR{S}{cond} Rd, Rn, op2`
Bitwise clear	`BIC{S}{cond} Rd, Rn, op2`
Bitwise or	`ORR{S}{cond} Rd, Rn, op2`

- Bitwise clear:
  - The `BIC` (Bit Clear) instruction performs an AND operation on the bits in Rn with the complements of the corresponding bits in the value of Operand2.
- Examples:
  - `Rn = 0xFF`
  - `op2 = 0b111, op2 = 0b1, op2 = 0b100`

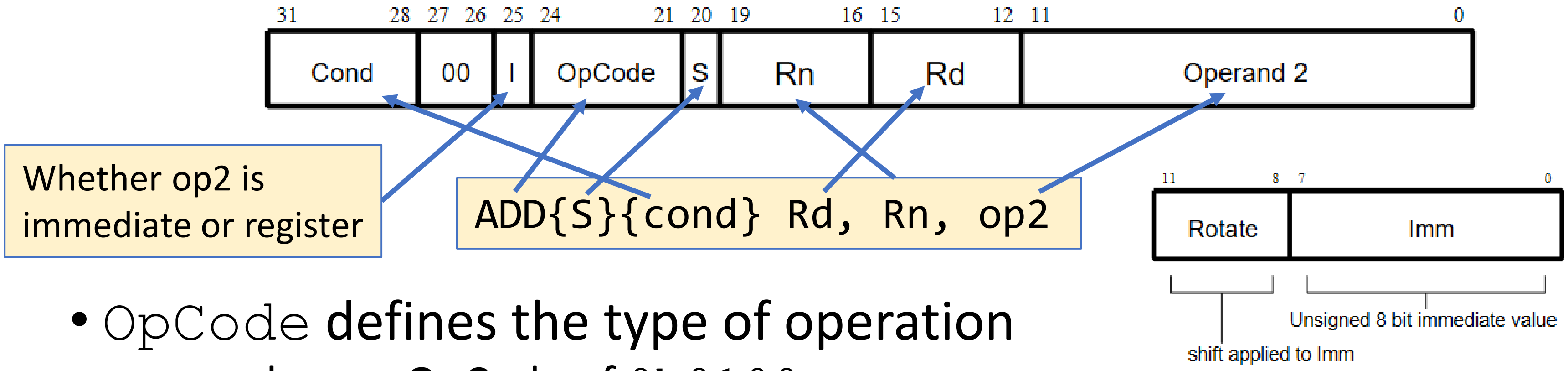
Logical Instructions

- These logical (including shift) instructions are very important when dealing with binary data.
- E.g. We want to get the header checksum of an IP package
  - Assume the data for the third line is `0x7644FA52` stored in `r0`.
  - Use `0xFFFF0000` as a mask, assume that we store it to `r1`.
  - Then use: `AND r2, r0, r1.`
  - `R2` will hold the checksum.
- How to get the protocol field?



Instruction Encoding

- Instructions like `ADD`, `SUB` all follow some form of encoding.
  - More examples will be revealed later in this module.
- Below is the encoding format shared by all data processing instructions, like `ADD`, `SUB`, `AND`, etc.:



31–28	27–26	25	24–21	20	19–16	15–12	11–0
Cond	00	I	OpCode	S	Rn	Rd	Operand 2

Whether op2 is immediate or register

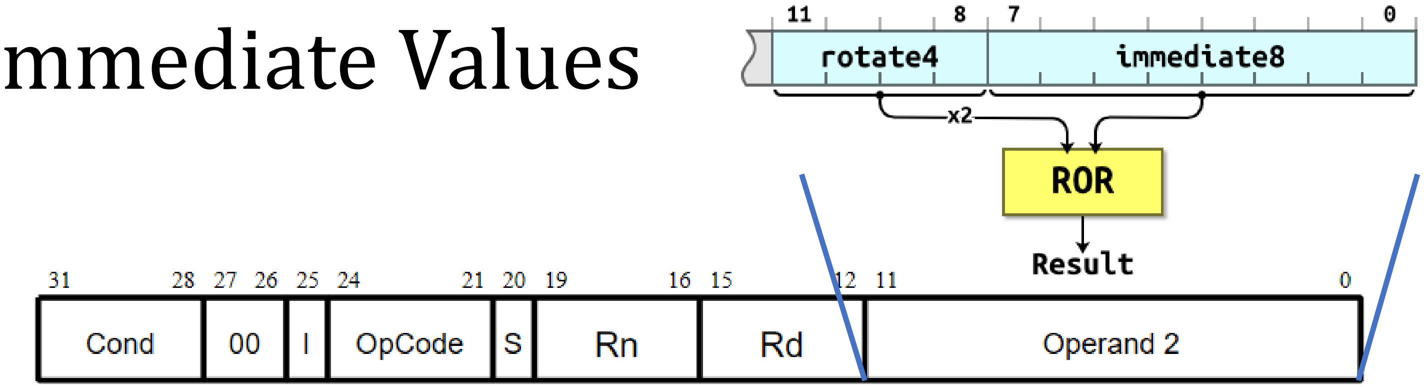
`ADD{S}{cond} Rd, Rn, op2`

11–8	7–0
Rotate	Imm

- `OpCode` defines the type of operation
  - `ADD` has an OpCode of `0b0100`.
  - `SUB` has an OpCode of `0b0010`

- `Rn`, `Rd` are registers, and are described by 4 bits
  - $2^4 = 16$ registers accessible.

Immediate Values



- The immediate (Imm) field is up to 8 bits long. Then further rotated right by a number indicated in the 4-bit rotate value.
  - The 4-bit rotate value stored in bits 11-8 is multiplied by two, giving a range of 0-30.

- Both of them are automatically generated based on your immediate value.
  - `ADD r1, r0, #0xFF0  (0b1111 1111 0000)`

    0b1111 1111 rotated right by 28 bits

  - `ADD r2, r0, #0x3FC00000 (0b 0011 1111 1100 0....)`

    0b1111 1111 rotated right by 10 bits

Immediate Values

Based on how these immediate values are represented in this encoding:

- Half of the numbers whose first 1 and last 1 being 8 bits apart can be represented
  - 0x8000007F: No
  - 0xC000003F: Yes
  - 0xE000001F: No

- All numbers whose first 1 and last 1 being less than 8 bits apart can be represented

- See the code example I uploaded, and carefully think about the data in the code.
  - The file is called: L2-immediates_value_rotation.s

Writing Assembly Code

- The instructions, directives, and pseudo-instructions must be preceded by a white space, either a tab or any number of spaces, even if you don’t have a label at the beginning.
  - Underlined terms will be covered later.

- Comments starts with a semi-colon (;)

- Saved code files should have an (.s) extension.
  - E.g., mycode.s rather than mycode.txt.
  - Although they are just text files.

C++ Code and Assembly

- Have a look at the C++ code and its corresponding assembly code I uploaded.

- Can you find the connections between these code lines?