UNIVERSITY OF TEXAS ARLINGTON

Computer Organization & Assembly Language Programming (CSE 2312)

Lecture 14: Midterm Review

Taylor Johnson

Announcements and Outline

- Quiz 4 after midterm
- Homework 4 due today
- Midterm 10/9
    - Chapter 1, 2 (ARM), Appendices A1-A6, Appendices B1-B2 (ARM)
    - 1 letter page cheat sheet, both sides
    - No calculator
- Midterm Review

What is this Course About?

- This course is about one fundamental question in computer science and engineering
- You probably do not yet know the answer
- ***How do computers compute?***
- What does the computer actually do when you ask it to do something (i.e., run a program you've written)?

Multilevel Architectures

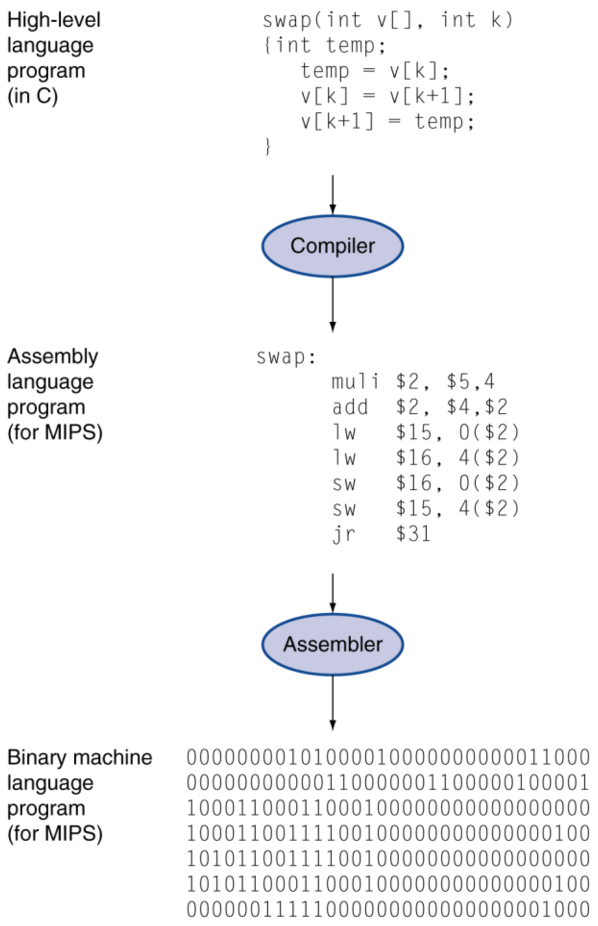
Level 4	Operating System Level	C / …
Level 3	Instruction Set Architecture (ISA) Level	Assembly / Machine Language
Level 2	Microarchitecture Level	n/a / Microcode
Level 1	Digital Logic Level	VHDL / Verilog
Level 0	Physical Device Level (Electronics)	n/a / Physics

Compilation vs. Interpretation

- Compilation:
  - your n-level program is translated into a program at a lower level
  - the program at the lower level is stored in memory, and executed
  - while running, the lower-level program controls the computer
- Interpretation:
  - An interpreter, implemented at a lower level, executes your n-level program line-by-line
  - The interpreter translates each line into lower-level code, and executes that code
  - The interpreter is the program that is running, not your code

Review: Levels of Program Code

- High-level language
  - Level of abstraction closer to problem domain
  - Provides for productivity and portability
- Assembly language
  - Textual representation of instructions
- Hardware representation
  - Binary digits (bits)
  - Encoded instructions and data

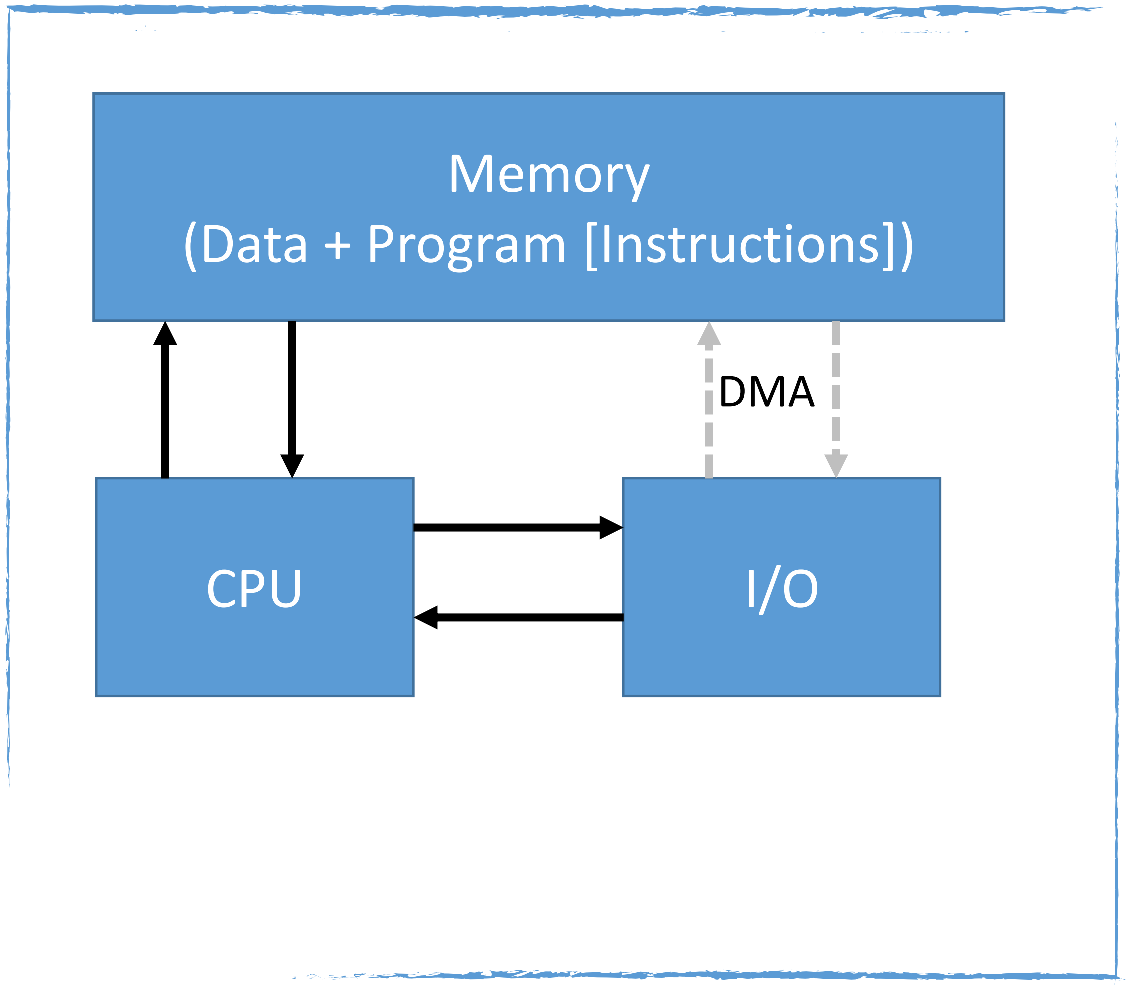

High-level language program (in C)
Compiler
Assembly language program (for MIPS)
Assembler
Binary machine language program (for MIPS)

```
swap(int v[], int k)
{int temp;
   temp = v[k];
   v[k] = v[k+1];
   v[k+1] = temp;
}
```

```
swap:
      muli $2, $5,4
      add  $2, $4,$2
      lw   $15, 0($2)
      lw   $16, 4($2)
      sw   $16, 0($2)
      sw   $15, 4($2)
      jr   $31
```

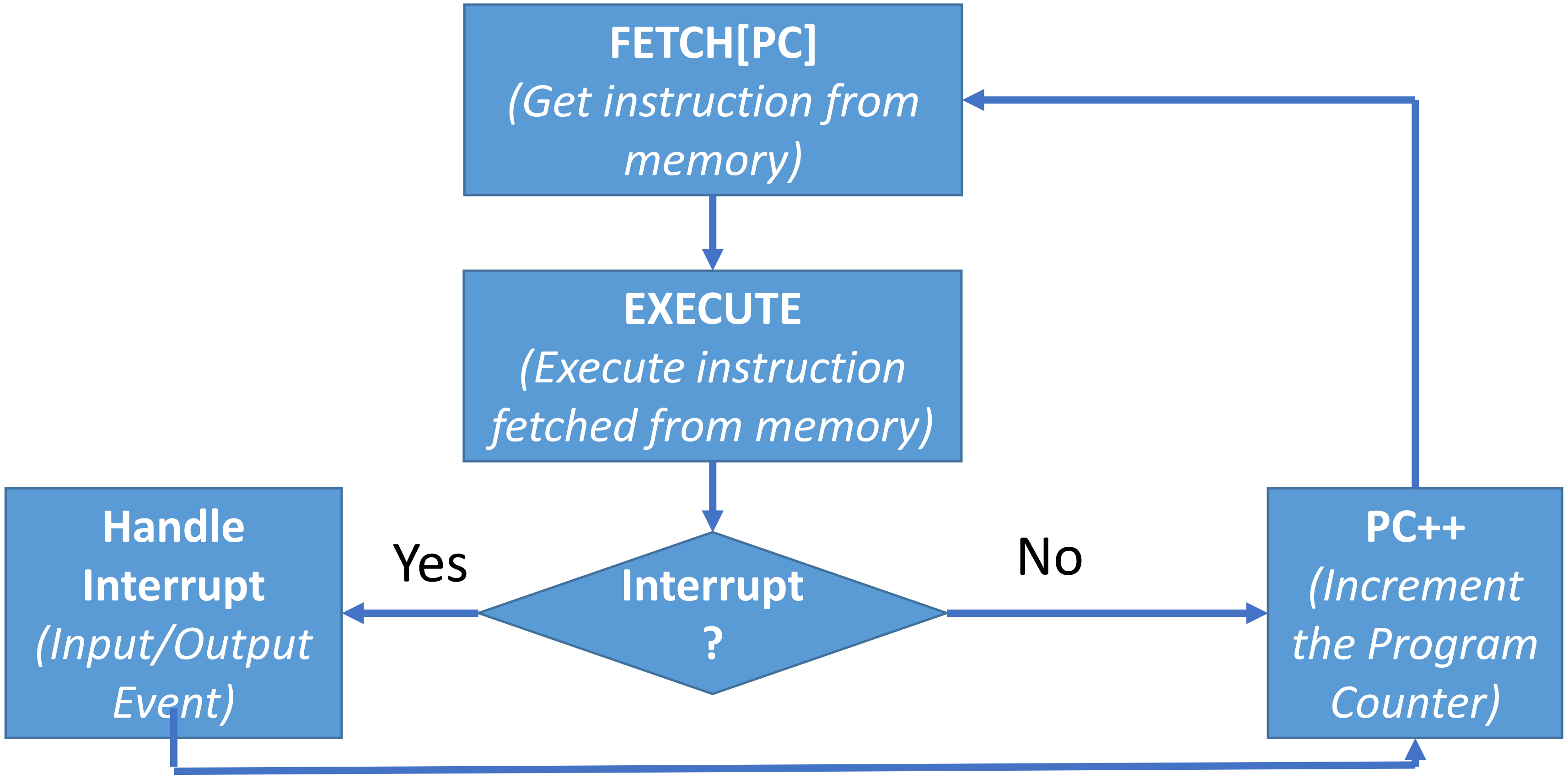
```
00000000101000010000000000011000
00000000000110000001100000100001
10001100011000100000000000000000
10001100111100100000000000000100
10101100111100100000000000000000
10101100011000100000000000000100
00000011111000000000000000001000
```

Von Neumann Architecture

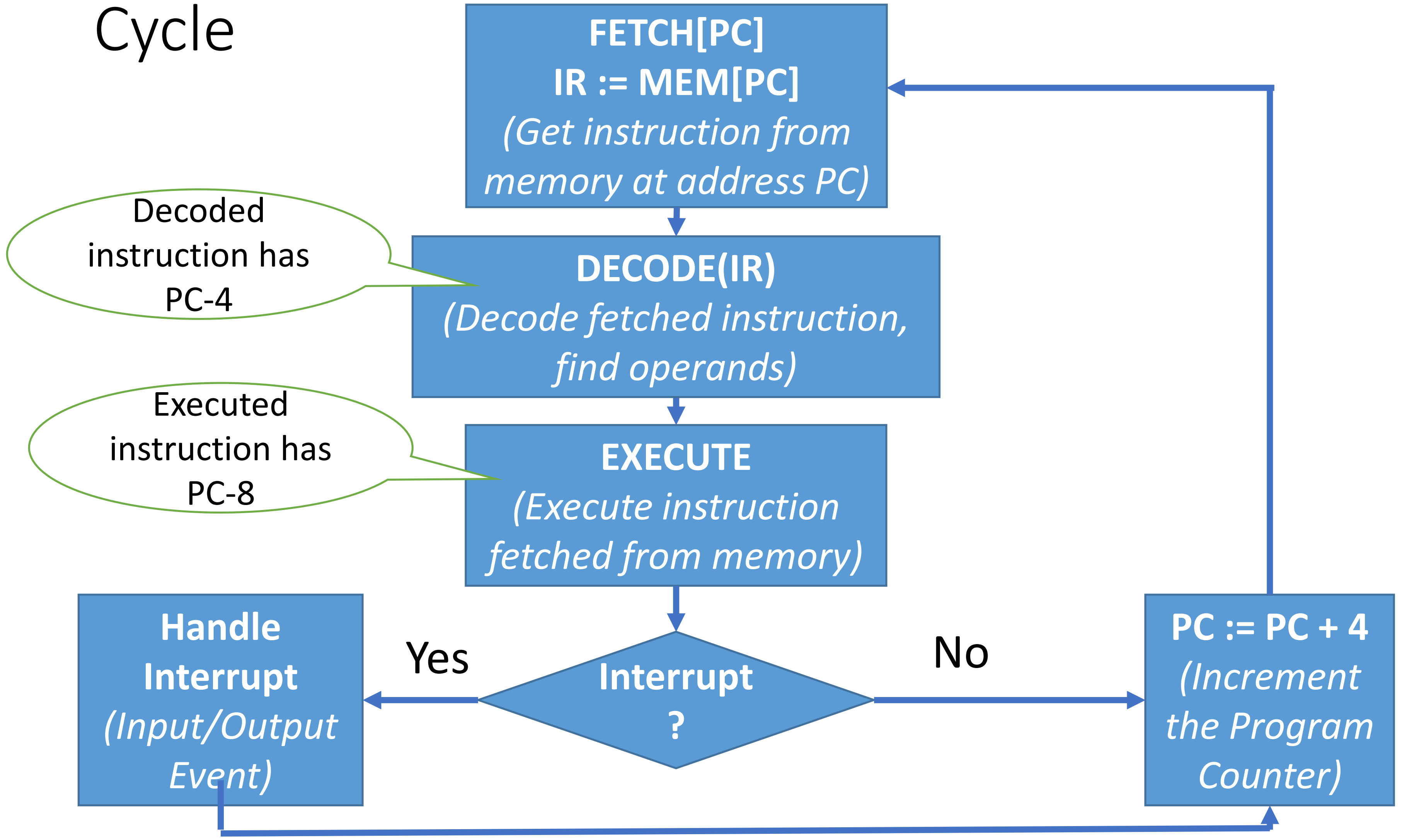


- Both data and program stored in memory
- Allows the computer to be “re-programmed”
- Input/output (I/O) goes through CPU
- I/O part is not representative of modern systems (direct memory access [DMA])
- Memory layout is representative of modern systems

Review: Abstract Processor Execution Cycle


FETCH[PC]
(Get instruction from memory)
EXECUTE
(Execute instruction fetched from memory)
Interrupt ?
Yes
No
Handle Interrupt
(Input/Output Event)
PC++
(Increment the Program Counter)

ARM 3-Stage Pipeline Processor Execution Cycle


FETCH[PC]
IR := MEM[PC]
(Get instruction from memory at address PC)
DECODE(IR)
(Decode fetched instruction, find operands)
EXECUTE
(Execute instruction fetched from memory)
Interrupt ?
Yes
No
Handle Interrupt
(Input/Output Event)
PC := PC + 4
(Increment the Program Counter)
Decoded instruction has PC-4
Executed instruction has PC-8

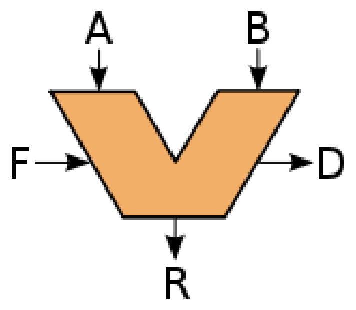
Some Processor Components

CPU

Register File

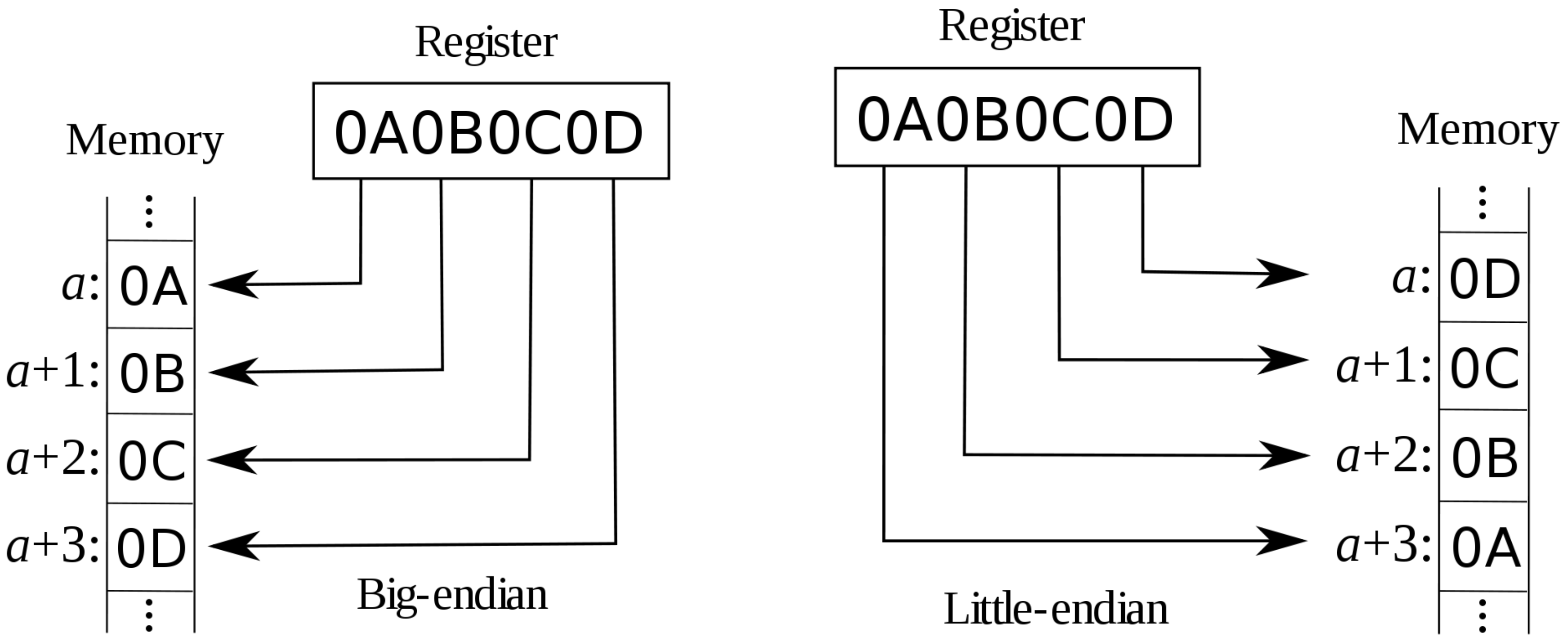
- Program Counter (PC)
- Instruction Register (IR)
- General Purpose Registers
  - Word size
  - Typically 16-32 of these
  - PC sometimes one of these
- Floating Point Registers

Arithmetic logic unit (ALU)



Floating Point Unit (FPU)

Endianness Example



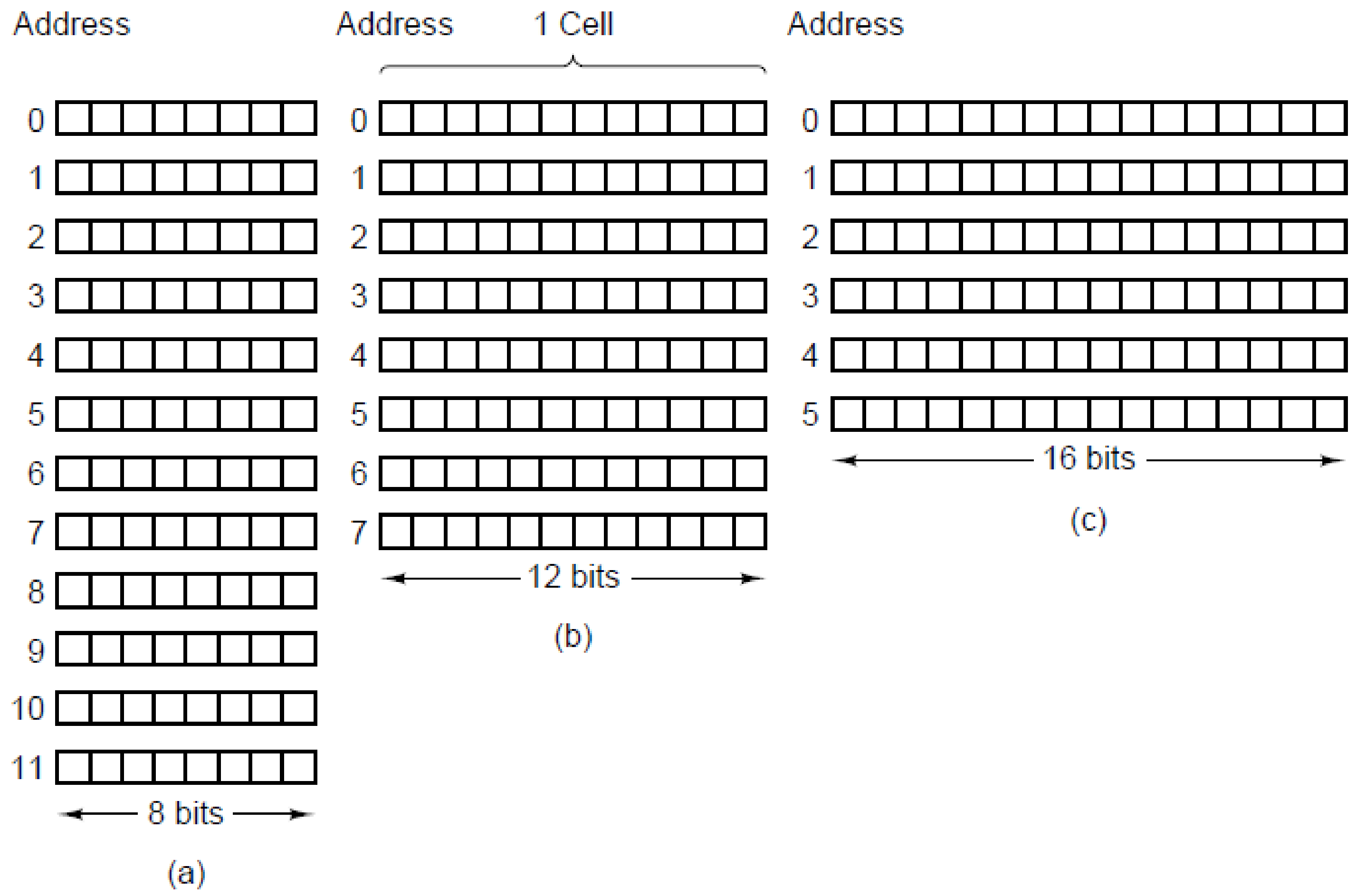
Memory: Words and Alignment

- Bytes are grouped into **words**
- Depending on the machine, a word can be:
  - 32 bits (4 bytes) , or
  - 64 bits (8 bytes), or … (16-bits, 128 bits, etc.)
- Oftentimes it is required that words are aligned
- This means that:
  - 4-byte words can only begin at memory addresses that are multiples of 4: 0, 4, 8, 12, 16…
  - 8-byte words can only begin at memory addresses that are multiples of 8: 0, 8, 16, 24, 32, …

Memory Cells and Addresses

- ***Memory cell***: a piece of memory that contains a specific number of bits
  - How many bits depends on the architecture
  - In modern architectures, it is almost universal that a cell contains *8 bits (1 byte)*, and that will be also our convention in this course
- ***Memory address***: a number specifying a location of a memory cell containing data
  - Essentially, a number specifying the location of a byte of memory

Cells and Addresses



Three ways of organizing a 96-bit memory

Memory as an Array

- Think of memory and addressing like you think of arrays

```
                  MEM[ADDR-1]     0x05
                  MEM[ADDR]       0xAB
                  MEM[ADDR+1]     0xF1
Suppose ADDR = 0x1000
                  MEM[0x0FFF]     0x05
                  MEM[0x1000]     0xAB
                  MEM[0x1001]     0xF1
                  MEM[...]        ...
```

How large is this memory?

Is this memory byte-addressable? How do you know how large (length and total memory size) an array is?

Hexadecimal

- Base 16
  - Compact representation of bit strings
  - 4 bits (also called a nibble or nybble) per hex digit

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

- Example: 0xECA8 6420
  - 1110 1100 1010 1000 0110 0100 0010 0000

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110

Binary-Decimal Conversions, Signed vs. Unsigned, Arithmetic Operations

- You should know how to convert binary/hex to/from decimal
- You should know how to represent signed numbers
  - Two's complement: changes sign of a number
  - Like complement (or putting a minus sign in front of a decimal number)
- You should know how to compute basic arithmetic and logical operations
  - AND, OR, NOT
  - ADD, SUB

Unsigned Binary Integers

- Given an n-bit number

$$x = x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: 0 to $+2^n - 1$
- Example
  - $0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 1011_2$
    $= 0 + \ldots + 1\times2^3 + 0\times2^2 + 1\times2^1 + 1\times2^0$
    $= 0 + \ldots + 8 + 0 + 2 + 1 = 11_{10}$
- Using 32 bits
  - 0 to +4,294,967,295

2s-Complement Signed Integers

- Given an n-bit number

$$x = -x_{n-1}2^{n-1} + x_{n-2}2^{n-2} + \cdots + x_1 2^1 + x_0 2^0$$

- Range: -2^{n-1} to $+2^{n-1} - 1$
- Example
  - $1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1100_2$
    $= -1\times2^{31} + 1\times2^{30} + \ldots + 1\times2^{2} + 0\times2^{1} + 0\times2^{0}$
    $= -2{,}147{,}483{,}648 + 2{,}147{,}483{,}644 = -4_{10}$
- Using 32 bits
  - –2,147,483,648 to +2,147,483,647

2s-Complement Signed Integers

- Bit 31 is sign bit
  - 1 for negative numbers
  - 0 for non-negative numbers
- –(–2n – 1) can’t be represented
- Non-negative numbers have the same unsigned and 2s-complement representation
- Some specific numbers

0:	0000 0000 … 0000
–1:	1111 1111 … 1111
Most-negative:	1000 0000 … 0000
Most-positive:	0111 1111 … 1111

Two's Complement Signed Negation

- Complement and add 1
  - Complement means 1 → 0, 0 → 1
  - Representation called one's complement

$$x + \bar{x} = 1111...111_2 = -1$$

$$\bar{x} + 1 = -x$$

- Example: negate +2
  - $+2 = 0000\ 0000\ \ldots\ 0010_2$
  - $-2 = 1111\ 1111\ \ldots\ 1101_2 + 1$
    $= 1111\ 1111\ \ldots\ 1110_2$

Units of Memory

- One bit (binary digit): the smallest amount of information that we can store:
  - Either a 1 or a 0
  - Sometimes refer to 1 as high/on/true, 0 as low/off/false
- One byte = 8 bits
  - Can store a number from 0 to 255
- Kilobyte (KB): $10^3 = 1000$ bytes
- Kibibyte (KiB): $2^{10} = 1024$ bytes
- Kilobit: (Kb): $10^3 = 1000$ bits (125 bytes)
- Kibibit: (Kib): $2^{10} = 1024$ bits (128 bytes)

Relative Performance

- Define Performance = 1/Execution Time
- "X is *n* time faster than Y"

$$\text{Performance}_X / \text{Performance}_Y$$
$$= \text{Execution time}_Y / \text{Execution time}_X = n$$

- Example: time taken to run a program
  - 10s on A, 15s on B
  - Execution Time$_B$ / Execution Time$_A$ = 15s / 10s = 1.5
  - So A is 1.5 times faster than B

CPU Time

$$\text{CPU Time} = \text{CPU Clock Cycles} \times \text{Clock Cycle Time}$$

$$= \frac{\text{CPU Clock Cycles}}{\text{Clock Rate}}$$

- Performance improved by
  - Reducing number of clock cycles
  - Increasing clock rate
  - Hardware designer must often trade off clock rate against cycle count

CPU Time Example

- Computer A: 2GHz clock, 10s CPU time
- Designing Computer B
  - Aim for 6s CPU time
  - Can do faster clock, but causes 1.2 × clock cycles
- How fast must Computer B clock be?

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A$$

$$= 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

Instruction Count and CPI

$$\text{Clock Cycles} = \text{Instruction Count} \times \text{Cycles per Instruction}$$

$$\text{CPU Time} = \text{Instruction Count} \times \text{CPI} \times \text{Clock Cycle Time}$$

$$= \frac{\text{Instruction Count} \times \text{CPI}}{\text{Clock Rate}}$$

- Instruction Count for a program = number of instructions in program
  - Determined by program, ISA and compiler
- Average cycles per instruction (CPI) = number of cycles to execute an instruction (on average)
  - Determined by CPU hardware
  - If different instructions have different CPI
    - Average CPI affected by instruction mix

UNIVERSITY OF TEXAS ARLINGTON

CPI Example

- Computer A: Cycle Time = 250ps, CPI = 2.0
- Computer B: Cycle Time = 500ps, CPI = 1.2
- Same ISA
- Which is faster, and by how much?

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$

$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps}$$

A is faster…

$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$

$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$

$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2$$

…by this much

CPI in More Detail

- If different instruction classes take different numbers of cycles

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

- Weighted average CPI

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \underbrace{\frac{\text{Instruction Count}_i}{\text{Instruction Count}}}_{\text{Relative frequency}} \right)$$

CPI Example

- Alternative compiled code sequences using instructions in classes A, B, C

Class	A	B	C
CPI for class	1	2	3
IC in sequence 1	2	1	2
IC in sequence 2	4	1	1

- Sequence 1: IC = 5
  - Clock Cycles
    = 2×1 + 1×2 + 2×3
    = 10
  - Avg. CPI = 10/5 = 2.0
- Sequence 2: IC = 6
  - Clock Cycles
    = 4×1 + 1×2 + 1×3
    = 9
  - Avg. CPI = 9/6 = 1.5

Performance Summary

$$\text{CPU Time} = \frac{\text{Instructions}}{\text{Program}} \times \frac{\text{Clock cycles}}{\text{Instruction}} \times \frac{\text{Seconds}}{\text{Clock cycle}}$$

- Performance depends on
  - Algorithm: affects IC, possibly CPI
  - Programming language: affects IC, CPI
  - Compiler: affects IC, CPI
  - Instruction set architecture: affects IC, CPI, T_c

Chapter 1 Summary

- Cost/performance is improving
  - Due to underlying technology development
- Hierarchical layers of abstraction
  - In both hardware and software
- Instruction set architecture
  - The hardware/software interface
- Execution time: the best performance measure
- Power is a limiting factor
  - Use parallelism to improve performance

ARM Arithmetic Instructions in Machine Language

Cond	F	I	Opcode	S	Rn	Rd	Operand2
1110	00	0	0100	0	0001	0101	0000 0000 0010
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits

- Example: `add r5, r1, r2`
- C equivalent: `r5 = r1 + r2`
- Machine language encoding above
- Opcode: 0100 means add (dependent on digital logic, some encoding)
- Rd: register destination operand. It gets the result of the operation
- Rn: first register source operand
- Operand2: second source operand
- I: Immediate. If I is 0, the second source operand is a register. If I is 1, the second source operand is a 12-bit immediate
- S: Set Condition Code
- Cond: Condition. Related to conditional branch instructions
- F: Instruction Format

Review: Operands Types

- Register operand: operand comes from the binary valued stored in a particular register in the CPU
  - Example: `add r0, r1, r2`
  - C code: `r0 = r1 + r2;`
- Immediate operand: operand value comes from instruction itself
  - Example: `add r0, r1, #1`
  - C code: `r0 = r1 + 1;`
- Memory operand: operand refers to memory
  - Example: `str r0, [r1]`
  - C code (roughly): `MEM[r1] = r0;`
  - Only for load / store instructions!
  - Several addressing modes (more on this later)

Review: Memory Operand Example 1

- C code:

```
g = h + A[8];
```

  - g in r1, h in r2, base address of A in r3
- Compiled ARM code:
  - Index 8 requires offset of 8 words
    - 4 bytes per word

```
@ load word
ldr r0, [r3, #32] @ r0 = MEM[r3 + 32]
add r1, r2, r0
```

base register

offset

Review: Memory Operand Example 2

- C code:

```
A[12] = h + A[8];
```

  - h in r2, base address of A in r3
- Compiled ARM code:
  - Index 8 requires offset of 32 (8 bytes, 4 bytes per word)

```
@ load word
ldr r0, [r3,#32] @ r0 = MEM[r3 + 32]
add r0, r2, r0
@ store word
str r0, [r3, #48] @ MEM[r3 + 48] = r0
```

Immediate/Literal Addressing

1. Immediate: ADD r2, r0, #5

cond	f	opcode	rn	rd	Immediate

- Operand comes from the instruction
- Example: 32-bit instruction to move 4 into R1
  - Result is R1 := 4

MOV R1 #4

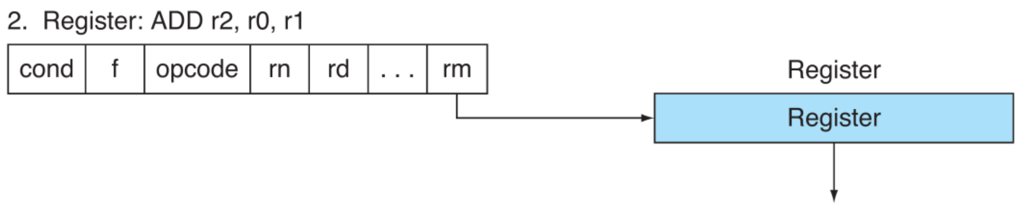
$$\underbrace{MOV}_{opcode\ field}\quad \overbrace{R1}^{destination\ register\ field}\quad \underbrace{\#4}_{immediate\ field}$$

- Useful for specifying small integer constants (avoids extra memory access)
- Can only specify small constants (limited by size of immediate field)
  - ARM: typically 8-12-bits

Register/Register-Direct Addressing

- Operand(s) come(s) from register(s)
  - Seen this many times already: ADD R0 R1 R2 does R0 := R1 + R2
  - Also: MOV R1 R2: the destination operand is specified by its register address (Result is R1 := R2)

$$\underbrace{MOV}_{\text{opcode field}} \quad \overbrace{R1}^{\text{destination register field}} \quad \underbrace{R2}_{\text{immediate field}}$$

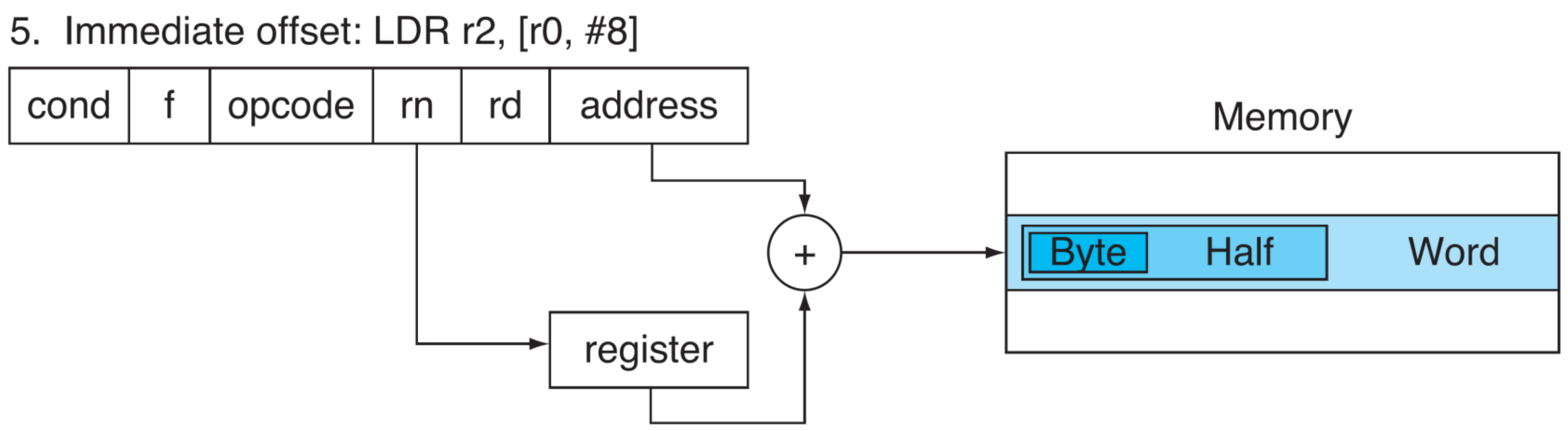


Register Indirect Addressing

- Operand comes from memory, with address specified by the value in a register (i.e., by a pointer) or by an immediate
- Example: ARM instruction to copy value from MEM[R4] into R1 (e.g., R1 := MEM[R4])

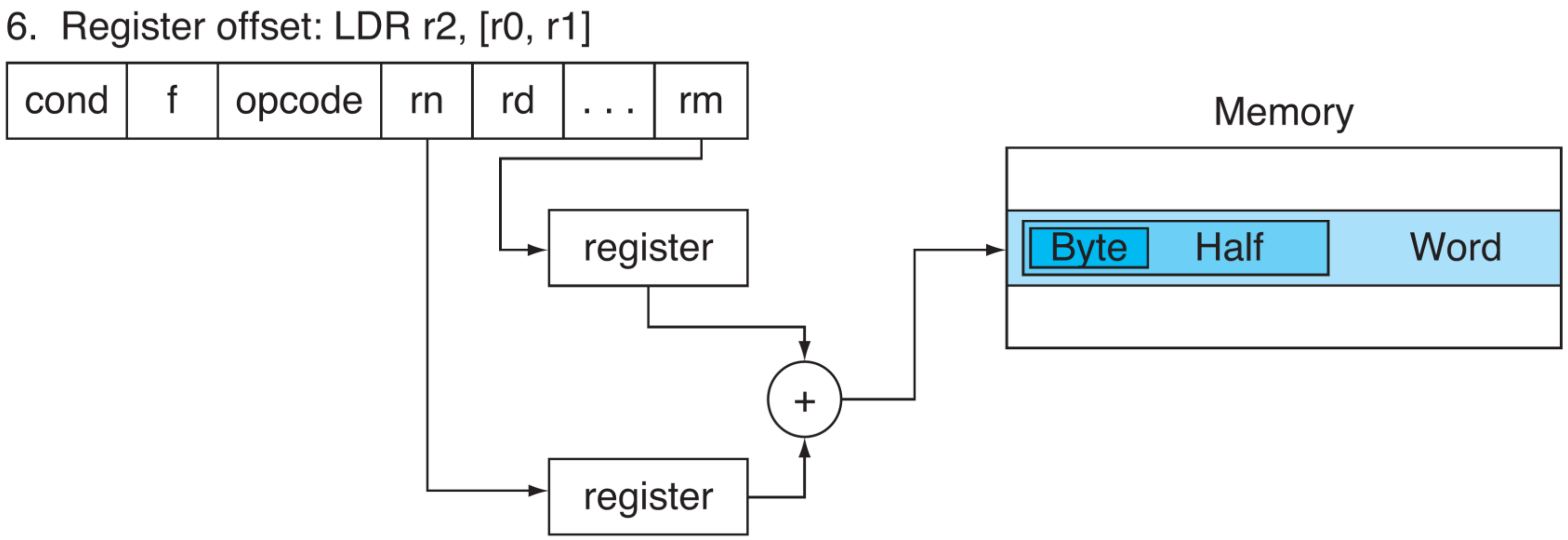
```
LDR R1 [R4]
```

Indirect with Immediate Offset



- Uses a register value and an immediate offset
- Example: LDR r2, [r0, #8]
  - Updates r2 = MEM[r0 + #8]

Indirect Register Offset



- Uses a register value and another register value as an offset
- Example: LDR r2, [r0, r1]
    - Updates r2 = MEM[r0 + r1]

Assembly Language Format

Label	Opcode	Operands	Comments
iloop:	add	r1,r1,#1	@ r1 := r1 + 1
	b	iloop	@ pc := iloop
val:	.byte	0x9F	@ put 0x96 at address val
s:	.asciz	“hello!”	@ put “hello!” at sequential addresses starting at address s

Assembly Instructions vs. Directives

- Instructions
    - Machine language equivalents
    - ADD, OR, NOT, LDR, STR, etc.
- Directives / Pseudoinstructions
    - Special commands given to the assembler that are converted to equivalent machine language during assembly process
    - Used for placing data in memory, etc.
    - .word, .byte, .asciz, .macro, .equ, etc.

Conditional Execution

- Current Program Status Register (CPSR)
  - Keeps track of arithmetic / logic (ALU) status
    - Example: last result was negative, zero, positive, had a carry, etc.
    - N (negative) / Z (zero) / C (carry) / V (overflow) bits
- Allows us to make conditional branches, etc. for conditional control flow changes (ifs, finite loops, etc.)
- Examples:

```
cmp r0, #0      @ compare r0 and #0
beq label       @ branch if r0 == 0
adds r0, r0, #1
bgt label       @ branch if r0 positive
```

ALU Status Bits

- N (negative) bit
  - Set to 1 when the result of the operation is negative, cleared to 0 otherwise
- Z (zero) bit
  - Set to 1 when the result of the operation is zero, cleared to 0 otherwise
- C (carry) bit
  - if the result of an addition is greater than or equal to 2^{32}
  - if the result of a subtraction is positive or zero
  - as the result of an inline barrel shifter operation in a move or logical instruction
- V (overflow) bit
  - Overflow occurs if the result of an add, subtract, or compare is greater than or equal to 2^{31}, or less than -2^{31}

Condition Code Suffixes

Suffix	Flags	Meaning
EQ	Z set	Equal
NE	Z clear	Not equal
CS or HS	C set	Higher or same (unsigned >=)
CC or LO	C clear	Lower (unsigned <)
MI	N set	Negative
PL	N clear	Positive or zero
VS	V set	Overflow
VC	V clear	No overflow
HI	C set and Z clear	Higher (unsigned >)
LS	C clear or Z set	Lower or same (unsigned <=)
GE	N and V the same	Signed >=
LT	N and V differ	Signed <
GT	Z clear, N and V the same	Signed >
LE	Z set, N and V differ	Signed <=
AL	Any	Always. This suffix is normally omitted.

Control Flow Translation Example

- Suppose our ISA does not have a multiply instruction
- How can we perform multiplication?
  - Create equivalent sequence of computations yielding the same result
  - Use addition, branch, comparisons, etc.
  - $A * B = \sum_{i=1}^{B} A = \underbrace{A + A + \cdots + A}_{B\ times}$
  - Example: $5 * 9 = \sum_{i=1}^{9} 5 = \underbrace{5 + 5 + \cdots + 5}_{9\ times} = 45$
- Generalizing: this is the basis of all our modern computations
- CPU does not have "visit website, buy shoes" instruction

Procedures: Iterative Multiply

- How do we write a loop?
- How does flow of control change with function (procedure) calls?

```
int multiply(int A, int B) {
    int product = 0;
    while (B > 0) {
        product += A;
        B--;
    }
    return product;
}
```

ARM Assembly for Iterative Multiply

```
.globl _start
_start:     mov    r0, #5       @ A = 5
            mov    r1, #3       @ B = 3
            bl     imul         @ call iterative multiply procedure

iloop:      b      iloop        @ infinite loop (for "termination")

imul:       mov    r2,#0        @ initialize result to 0
imul_loop:  cmp    r0,#0        @ r0 == 0?
            beq    imul_done    @ if r0 == 0, set PC = imul_done
            add    r2,r2,r1     @ r2 += r1
            sub    r0,r0,#1     @ r0 -= 1
            b      imul_loop    @ branch to imul_loop
imul_done:  mov    r0,r2        @ r0 = r2
            bx     lr           @ set PC = LR
```

Summary of Caller and Callee Steps

- Caller steps:
  - Step 1: Put arguments in the registers r0, r1, r2, r3.
  - Step 2: Branch to the function, using the bl instruction.
  - Step 3: After the function has returned, recover the return value (if any), and use it.
- Callee (called function) steps:
  - Step 1 (preamble): Allocate memory on the stack, and save register rl, and other registers that the function modifies, to the stack.
  - Step 2: Do the main body of the function.
  - Step 3 (wrap-up):
    - Store the return value (if any) on r0, second return value (if any) on r1.
    - Restore, from the stack, the original values of all registers that the function modified, as well as the value of register lr.
    - Deallocate memory on the stack (increment sp).
    - Branch to the return address using instruction bx.

The Stack

- Last-in, first-out (LIFO) data structure
  - Last data put in comes out first
  - Common analogy: like a quarter / coin holder in your car, the last coin put in comes out first
- Stack pointer (SP) register: points to current address of stack (i.e., the last thing in)
  - **YOU** must initialize it! Typically use address 0x100000
  - `mov sp, #0x100000`
- Stack instructions
  - `PUSH {r0}` means:
    - `SUB sp, sp, #4`
    - `STR r0, [sp]`
  - `POP {r0}` means:
    - `LDR r0, [sp]`
    - `ADD sp, sp, #4`
  - Can use lists of registers, e.g., `PUSH {r0,r1}` is:

```
SUB sp, sp, #8
STR r0, [sp]
STR r1, [sp,#4]
```

Basic Function Call Example

```
int ex(int g, int h, int i, int j) {
    int f;
    f = (g + h) - (i + j);
    return f;
}

r0 = g, r1 = h, r2 = I, r3 = j, r4 = f
```

Basic Function Call Example Assembly

```
ex:                  ; label for function name
SUB sp, sp, #12      ; adjust stack to make room for 3 items
STR r6, [sp,#8]      ; save register r6 for use afterwards
STR r5, [sp,#4]      ; save register r5 for use afterwards
STR r4, [sp,#0]      ; save register r4 for use afterwards

ADD r5,r0,r1        ; register r5 contains g + h
ADD r6,r2,r3        ; register r6 contains i + j
SUB r4,r5,r6        ; f gets r5 – r6, ie: (g + h) – (i + j)
MOV r0,r4           ; returns f (r0 = r4)

LDR r4, [sp,#0] ; restore register r4 for caller
LDR r5, [sp,#4] ; restore register r5 for caller
LDR r6, [sp,#8] ; restore register r6 for caller
ADD sp,sp,#12   ; adjust stack to delete 3 items
MOV pc, lr      ; jump back to calling routine
```

Basic Function Call Example Stack

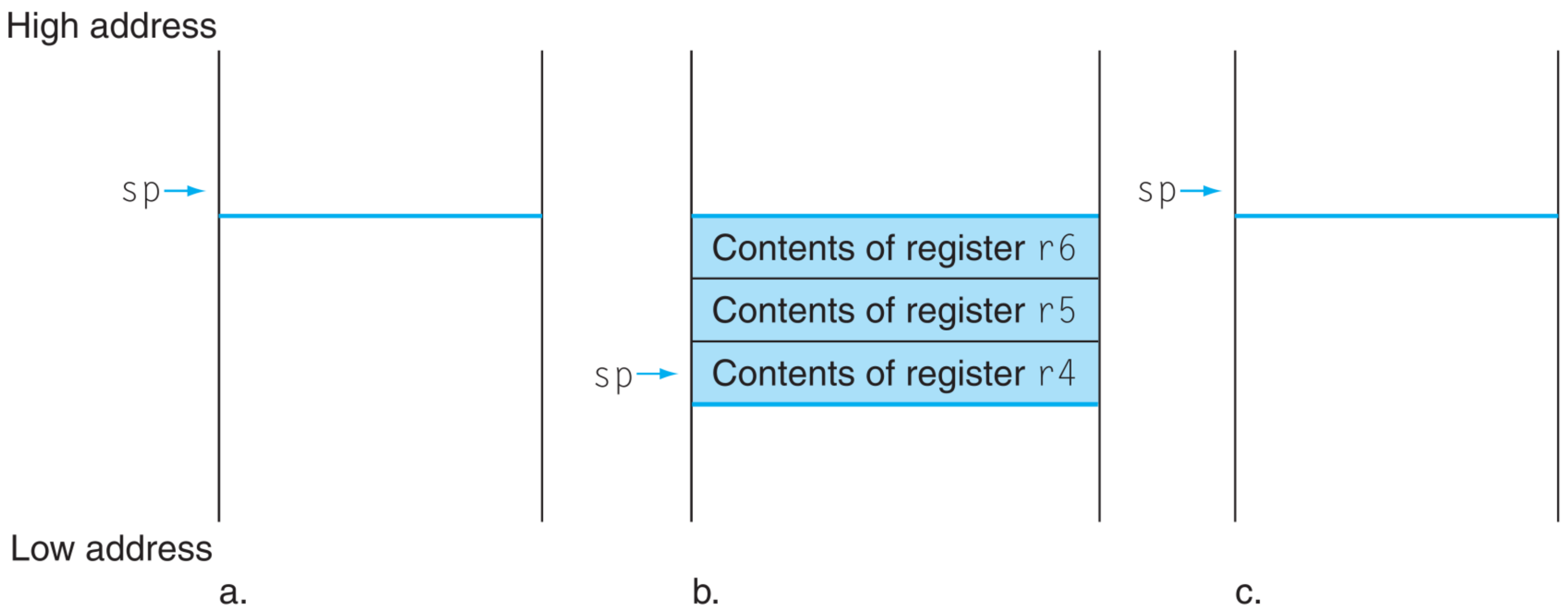


FIGURE 2.10 The values of the stack pointer and the stack (a) before, (b) during, and (c) after the procedure call. The stack pointer always points to the "top" of the stack, or the last word in the stack in this drawing.

Basic Function Call Example Assembly (Push/Pop)

```
ex:                 ; label for function name
PUSH {r4,r5,r6}   ; save r4, r5, r6, decrement sp by 12

ADD r5,r0,r1    ; register r5 contains g + h
ADD r6,r2,r3    ; register r6 contains i + j
SUB r4,r5,r6    ; f gets r5 – r6, ie: (g + h) – (i + j)
MOV r0,r4       ; returns f (r0 = r4)

POP {r4,r5,r6} ; restore r4, r5, r6, increment sp by 12
MOV pc, lr      ; jump back to calling routine
```

State Preservation Across Procedure Calls

Preserved	Not preserved
Variable registers: `r4-r11`	Argument registers: `r0-r3`
Stack pointer register: `sp`	Intra-procedure-call scatch register: `r12`
Link register: `lr`	Stack below the stack pointer
Stack above the stack pointer	

Recursive Function Example: Factorial

- How do we write function factorial in C, as a recursive function?

```
int factorial(int N)
{
  if (N== 0) return 1;
  return N* factorial(N -1);
}
```

- How do we write function factorial in assembly?

```
@ factorial main body
mov r4, r0
cmp r4, #0
moveq r0, #1
beq factorial_exit

sub r0, r4, #1
bl factorial
mov r5, r0
mul r0, r5, r4
```

UNIVERSITY OF TEXAS ARLINGTON

Recursive Function Example: Factorial

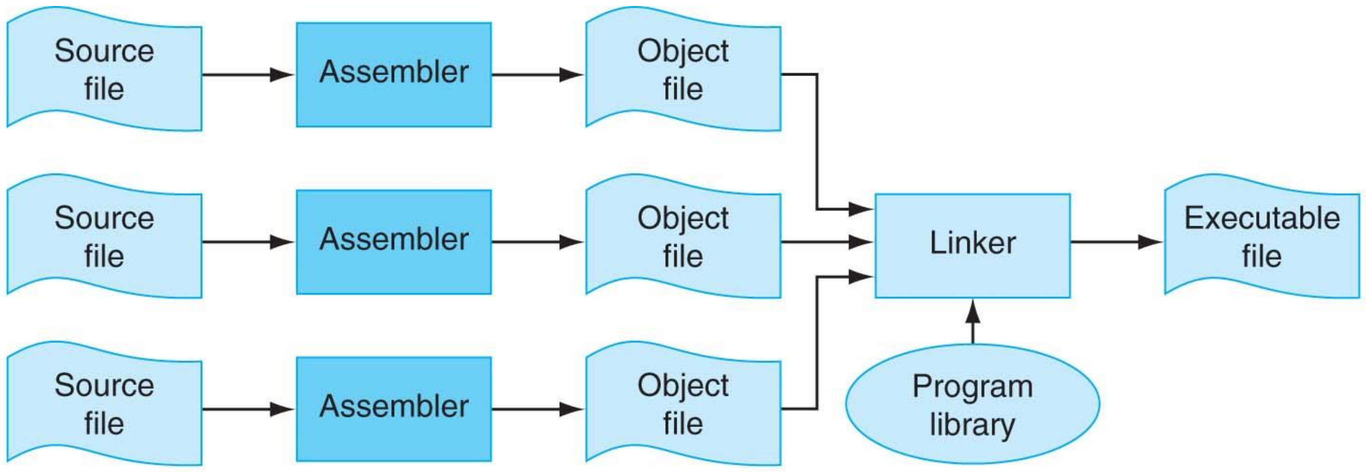
```
      @ factorial preamble
fact: push {r4,r5,lr}

      @ factorial body
      mov r4, r0
      cmp r4, #0
      moveq r0, #1
      beq fact_exit

      sub r0, r4, #1
      bl fact
      mov r5, r0
      mul r0, r5, r4
```

```
@ factorial wrap-up
fact_exit:
   pop {r4,r5,lr}
   bx lr
```

Assembly Process

- Insuffiency of one pass
  - Suppose we have labels (symbols).
  - How do we calculate the addresses of labels later in the program?
  - Example:
    - `ADDR: 0x1000          b` **`done`**
    - …   `// Other instructions and data`
    - `ADDR: 0x????   `**`done:`**` add r1, r2, r0`
    - …
    - How to compute address of label **`done`**?
- Two-Pass Assemblers
  - First Pass: iterate over instructions, build a symbol table, opcode table, expand macros, etc.
  - Second Pass: iterate over instructions, printing equivalent machine language, plugging in values for labels using symbol table

Assembly Process



The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

Linking and Loading

- Linking: combining multiple program modules (pieces of code) into executable program
  - Examples: using our _tests files to load inputs to your programs, calling library functions like printf, etc.
- Loading: getting executable running on machine
  - Examples: calling QEMU with our binary
- Static linking
  - Combine multiple object files into single binary
- Dynamic linking
  - Load library shared code at runtime
  - Not talking about this: operating system concept
  - Examples: Windows DLLs

Review: Array Example

```
.globl _start
_start: mov        r1,#0 @ r1 := 0
        ldr    r0,=arrayPtr       @ r0 := arrayPtr
        ldr    r3,=arrayEnd       @ r3 := arrayEnd
        ldrb   r4,[r3,#0]  @ r4 := MEM[R3 + 0]
loop:   ldrb   r2,[r0,#0]  @ r3 := MEM[r0]
        cmp    r2,r4       @ r0 == 0xFF ?
        beq    done        @ branch if done
        add    r1,r1,r2    @ r1 := r1 + r2
        add    r0,r0,#1    @ r0 := r0 + #1
        b      loop        @ pc = loop (address)
done:   strb   r1,[r2]     @ MEM[r2] := r1
iloop:         b      iloop @ infinite loop
```

```
arrayPtr:
        .byte 2
        .byte 3
        .byte 5
        .byte 7
        .byte 11
        .byte 13
        .byte 17
        .byte 19
        .byte 23
        .byte 29
        .byte 31
        .byte 37
        .byte 41
        .byte 43
        .byte 47
arrayEnd:
        .byte 0xFF
```

Review: Macros

- Another assembler directive
  - Like .byte, .word, .asciz, that we've seen a little of before
- Way to refer to commonly used or repeated code
- Similar to an assembly procedure or function, ***but expanded (evaluated) at assembly time***, not run time
- Similar to #define in C, which is replaced by compiler at compile time
- Macro call: use of macro as an instruction
- Macro expansion: replacement of macro body by the corresponding instructions

Review: Macro Example

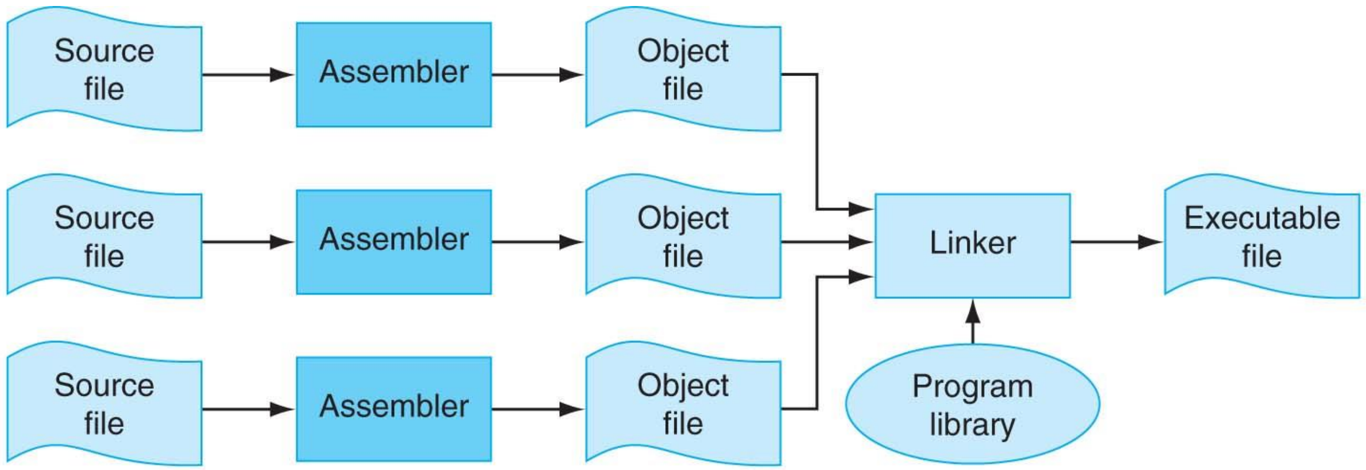
```
.globl _start
_start: .macro addVals adA, adB
        ldrb    r2,[\adA]       @ r2 := MEM[adA]
        ldrb    r3,[\adB]       @ r3 := MEM[adB]
        sub     r5,r2,r3        @ r5 := r2 - r3 = A - B
        strb    r5,[\adA]       @ MEM[adA] = r5
        ldrb    r2,[\adA]       @ r2 := MEM[adA]
        add     \adA,\adA,#1    @ r0 := r0 + 1
        add     \adB,\adB,#1    @ r1 := r1 + 1
        .endm                   @ end macro definition
init:   ldr     r0,=A           @ r0 := A (address)
        ldr     r1,=B           @ r1 := B (address)
        ldr     r4,=A_end       @ r4 := A_end (address)
        addVals         r0,r1   @ call macro
done:   b       done            @ infinite loop
```

```
A:      .byte 9, 8, 7, 6
A_end:       .byte 0
B:      .byte 1, 1, 1, 1
B_end:       .byte 0
```

Review: Assembly Process

- Insuffiency of one pass
  - Suppose we have labels (symbols).
  - How do we calculate the addresses of labels later in the program?
  - Example:
    - `ADDR: 0x1000           b` **`done`**
    - `…                 // Other instructions and data`
    - `ADDR: 0x????   ` **`done:`** `add r1, r2, r0`
    - …
    - How to compute address of label **`done`**?
- Two-Pass Assemblers
  - First Pass: iterate over instructions, build a symbol table, opcode table, expand macros, etc.
  - Second Pass: iterate over instructions, printing equivalent machine language, plugging in values for labels using symbol table

Review: Assembly Process



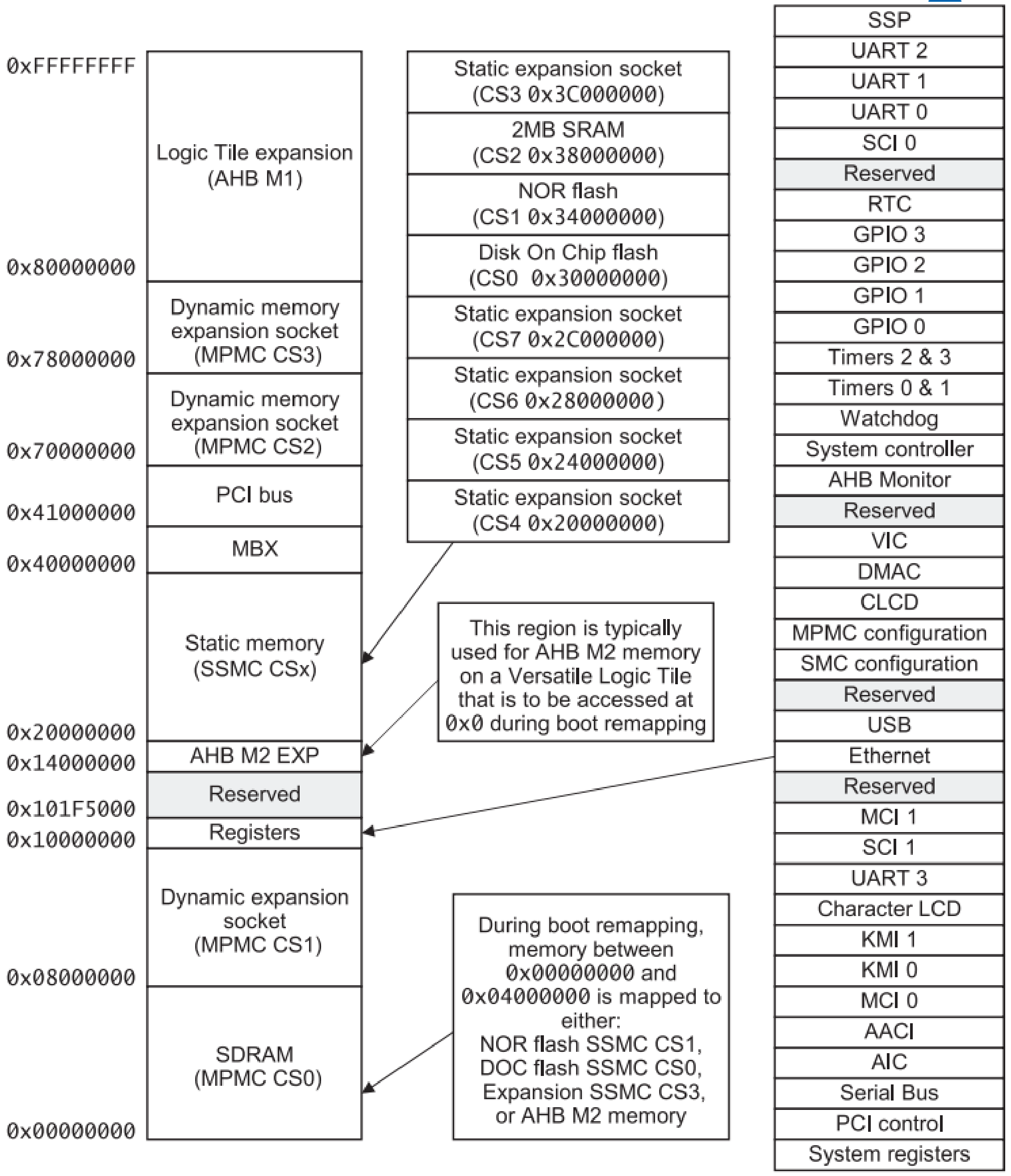
The process that produces an executable file. An assembler translates a file of assembly language into an object file, which is linked with other files and libraries into an executable file.

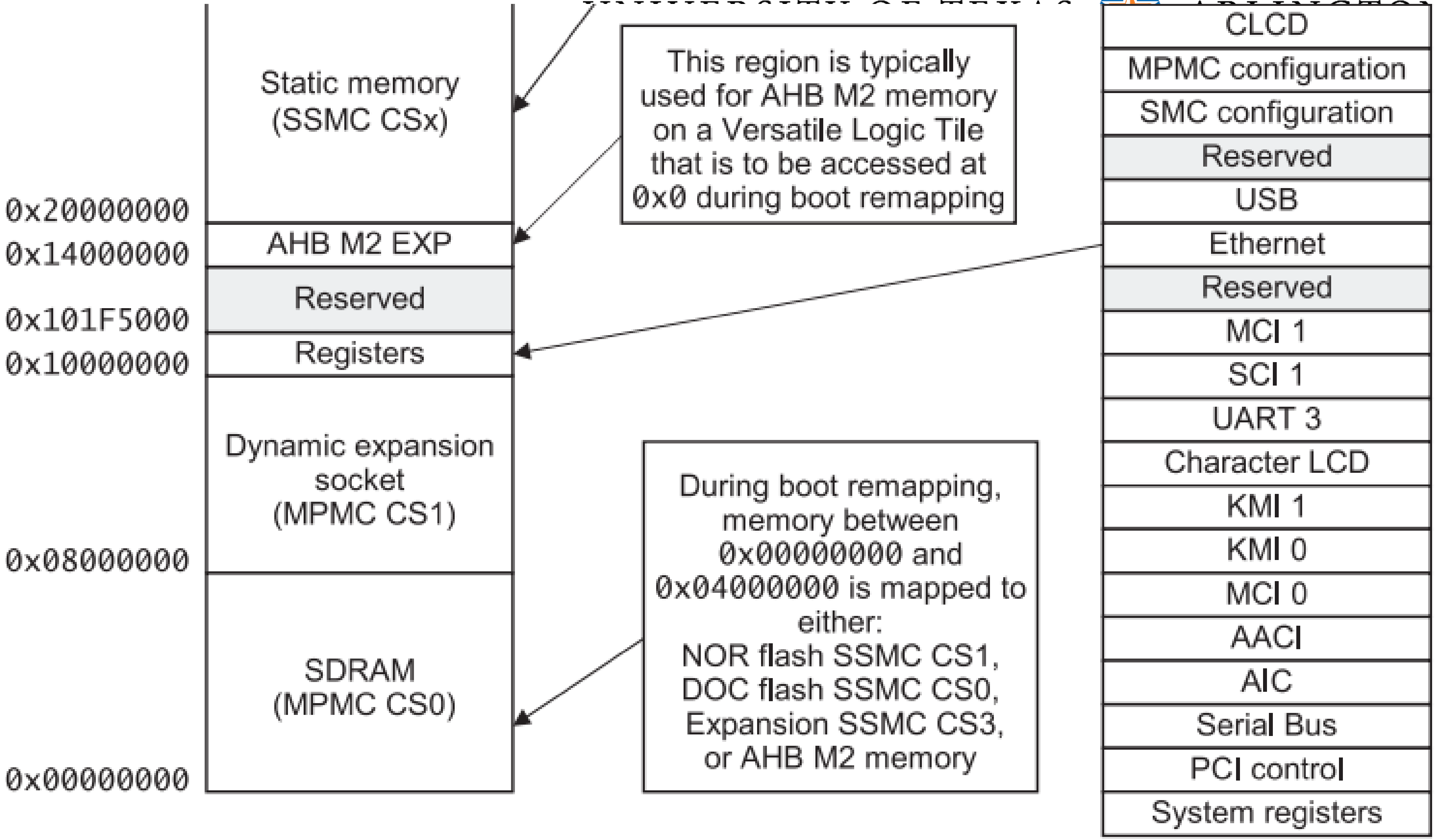
Review: Memory-Mapped I/O Example

- Some of our original examples displayed output to console by writing to a special memory address

```
.equ        ADDR_UART0, 0x101f1000
ldr         r0,=ADDR_UART0 @ r0 := 0x 101f 1000
mov         r2,#0x0D                @ R2 := 0x0D (return \r)
str         r2,[r0]                 @ MEM[r0] := r2
```

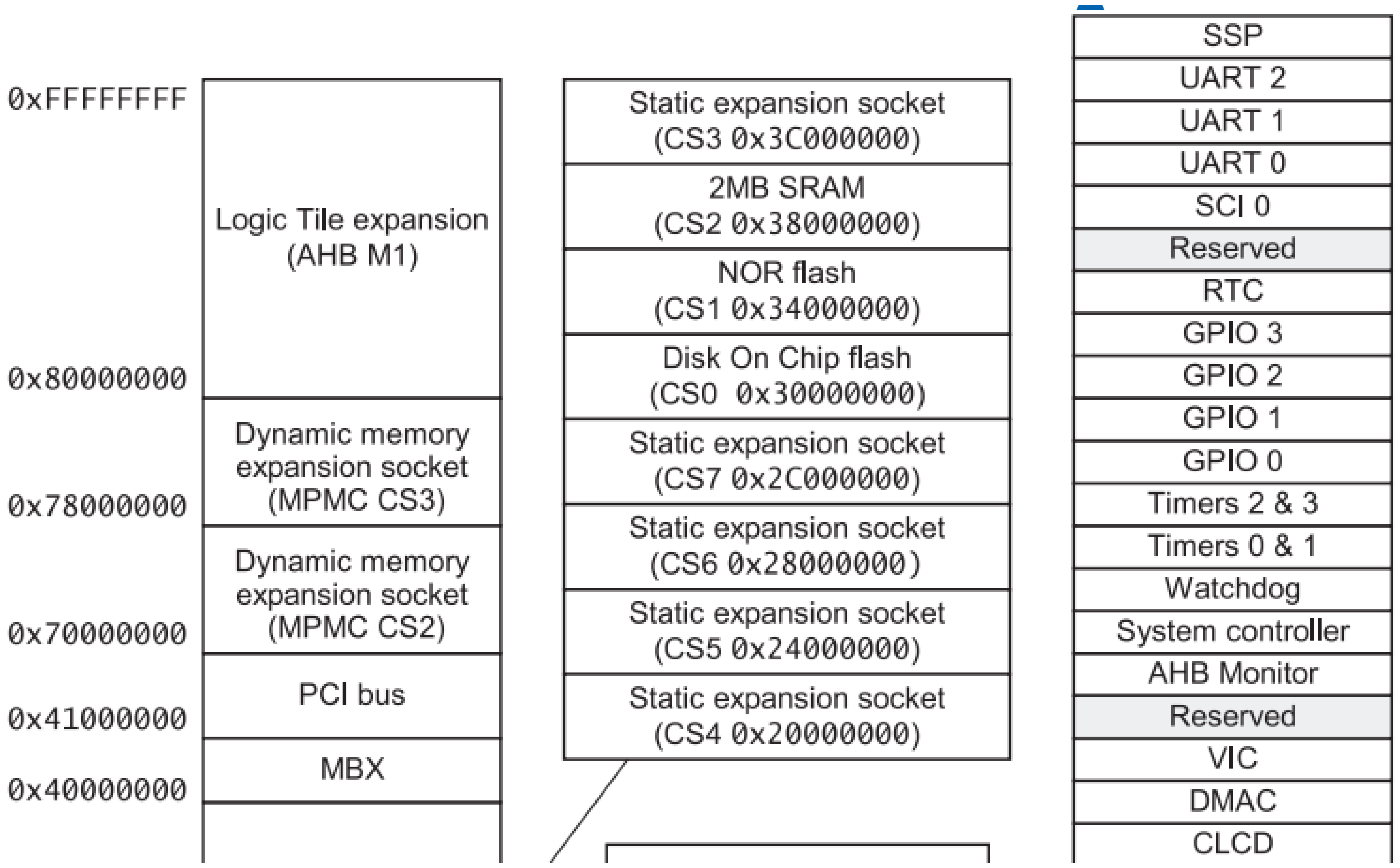
- How does this work?
  - Registers on peripheral devices (keyboards, monitors, network controllers, etc.) are addressable in same address space as main memory


0xFFFFFFFF
0x80000000
0x78000000
0x70000000
0x41000000
0x40000000
0x20000000
0x14000000
0x101F5000
0x10000000
0x08000000
0x00000000
Logic Tile expansion (AHB M1)
Dynamic memory expansion socket (MPMC CS3)
Dynamic memory expansion socket (MPMC CS2)
PCI bus
MBX
Static memory (SSMC CSx)
AHB M2 EXP
Reserved
Registers
Dynamic expansion socket (MPMC CS1)
SDRAM (MPMC CS0)
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping
During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either: NOR flash SSMC CS1, DOC flash SSMC CS0, Expansion SSMC CS3, or AHB M2 memory
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD
MPMC configuration
SMC configuration
Reserved
USB
Ethernet
Reserved
MCI 1
SCI 1
UART 3
Character LCD
KMI 1
KMI 0
MCI 0
AACI
AIC
Serial Bus
PCI control
System registers

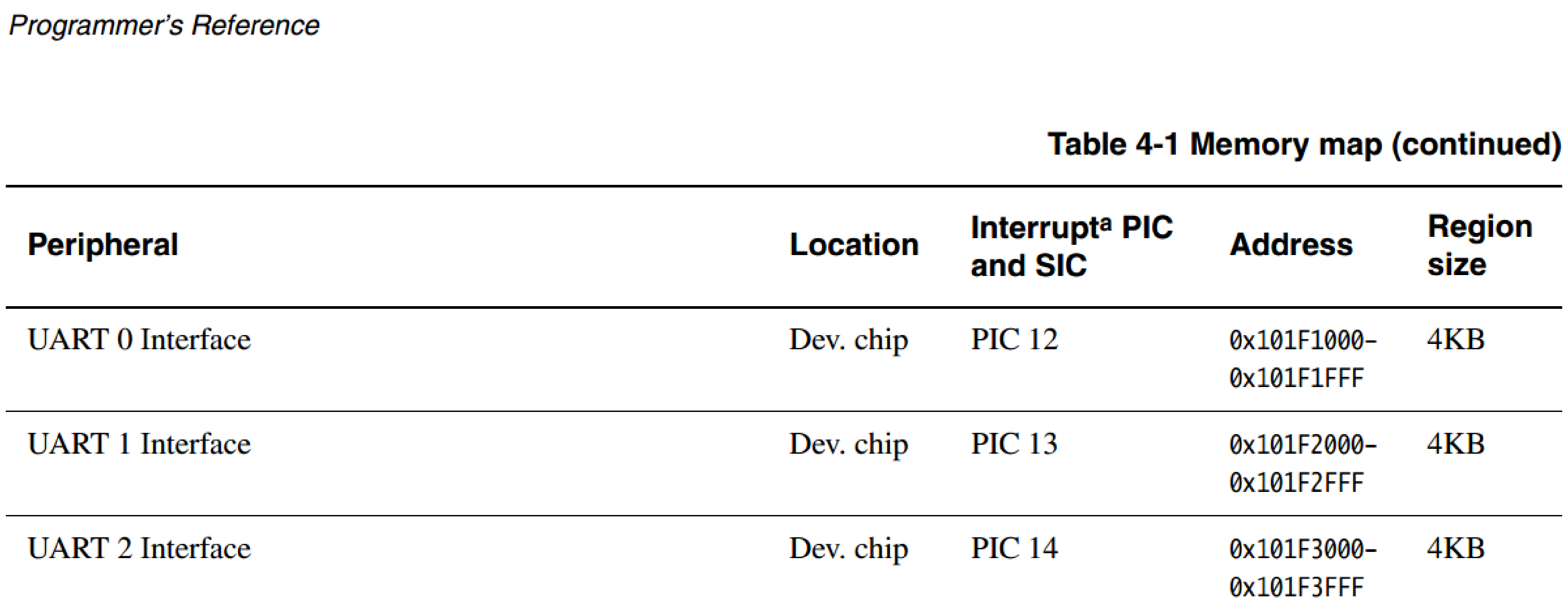


This region is typically used for AHB M2 memory on a Versatile Logic Tile that is to be accessed at 0x0 during boot remapping

During boot remapping, memory between 0x00000000 and 0x04000000 is mapped to either:
NOR flash SSMC CS1,
DOC flash SSMC CS0,
Expansion SSMC CS3,
or AHB M2 memory


0xFFFFFFFF
0x80000000
0x78000000
0x70000000
0x41000000
0x40000000
Logic Tile expansion (AHB M1)
Dynamic memory expansion socket (MPMC CS3)
Dynamic memory expansion socket (MPMC CS2)
PCI bus
MBX
Static expansion socket (CS3 0x3C000000)
2MB SRAM (CS2 0x38000000)
NOR flash (CS1 0x34000000)
Disk On Chip flash (CS0 0x30000000)
Static expansion socket (CS7 0x2C000000)
Static expansion socket (CS6 0x28000000)
Static expansion socket (CS5 0x24000000)
Static expansion socket (CS4 0x20000000)
SSP
UART 2
UART 1
UART 0
SCI 0
Reserved
RTC
GPIO 3
GPIO 2
GPIO 1
GPIO 0
Timers 2 & 3
Timers 0 & 1
Watchdog
System controller
AHB Monitor
Reserved
VIC
DMAC
CLCD

Address from Memory-Map in Manual



Programmer's Reference

Table 4-1 Memory map (continued)

Peripheral	Location	Interrupt[a] PIC and SIC	Address	Region size
UART 0 Interface	Dev. chip	PIC 12	0x101F1000-0x101F1FFF	4KB
UART 1 Interface	Dev. chip	PIC 13	0x101F2000-0x101F2FFF	4KB
UART 2 Interface	Dev. chip	PIC 14	0x101F3000-0x101F3FFF	4KB

http://infocenter.arm.com/help/topic/com.arm.doc.dui0224i/DUI0224I_realview_platform_baseboard_for_arm926ej_s_ug.pdf

Review: ELF Header Example

```
$ arm-none-eabi-objdump -f example.elf

example.elf:     file format elf32-littlearm
architecture: arm, flags 0x00000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x00010000
```

Review: ELF Symbol Table Example

```
$ arm-none-eabi-objdump -t example.elf
example.elf:     file format elf32-
littlearm
SYMBOL TABLE:
00010000 l    d  .text  00000000 .text
00010028 l       .text  00000000 rfib
00010024 l       .text  00000000 iloop
0001004c l       .text  00000000 rfib_exit
0001005c g       .text  00000000 _tests
00010000 g       .text  00000000 _start
```

local

global

Program starts at this address

Loading

- Get the binary loaded into memory and running
- More an operating systems concept
  - E.g., load an executable into memory and start it
  - Handled by QEMU for our purposes
    - Loads our binary starting at a particular memory address (`0x10000`)
    - Code at low, initial address (~`0x00000`) branches to that address

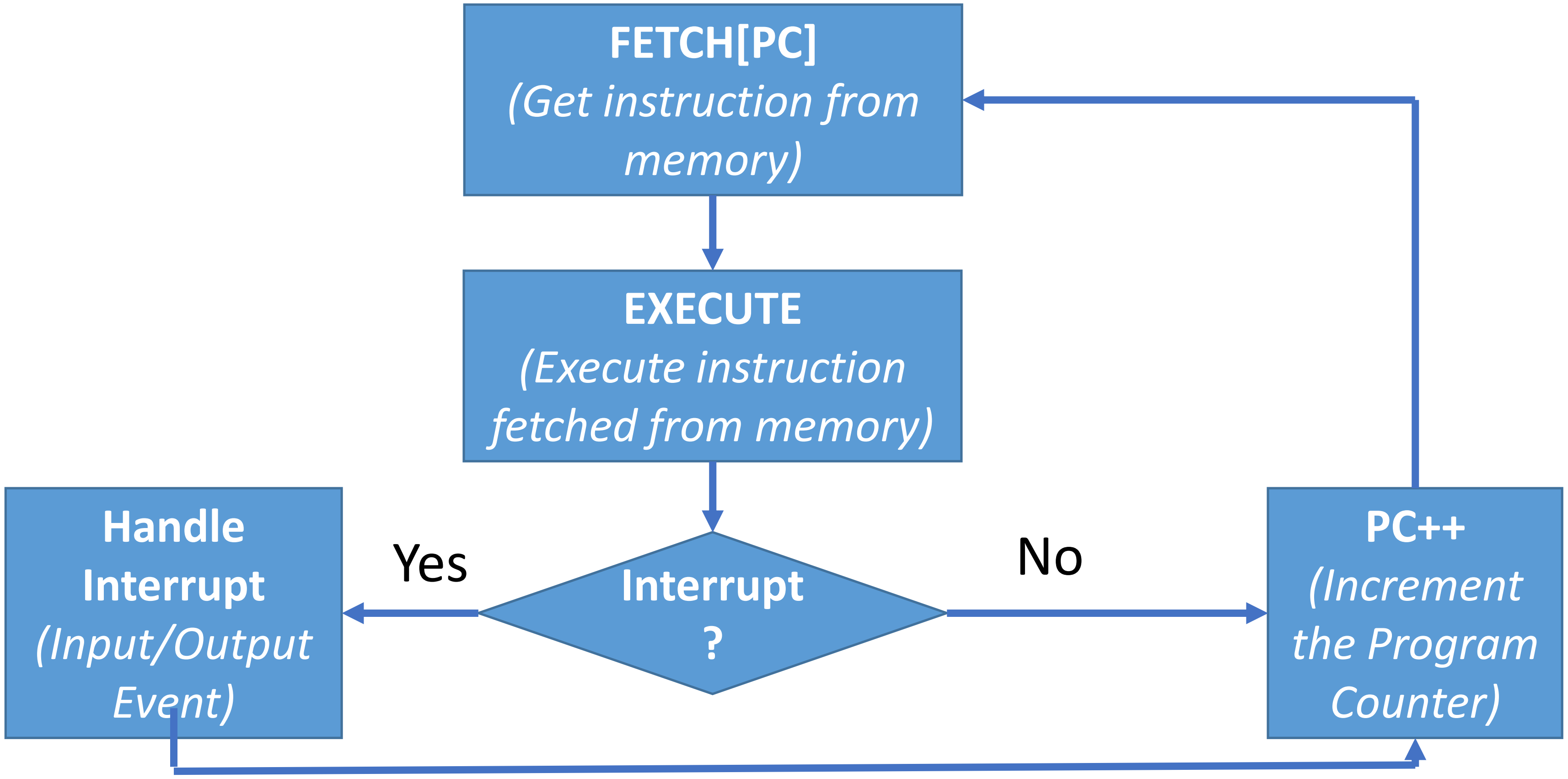
```
0x00000000:   e3a00000      mov  r0, #0     ; 0x0
0x00000004:   e59f1004      ldr  r1, [pc, #4]     ; 0x10
0x00000008:   e59f2004      ldr  r2, [pc, #4]     ; 0x14
0x0000000c:   e59ff004      ldr  pc, [pc, #4]     ; 0x18
0x00000010:   00000183
0x00000014:   0x000100
0x00000018:   0x010000     ; offset!
```

ARM 3 Stage Pipeline

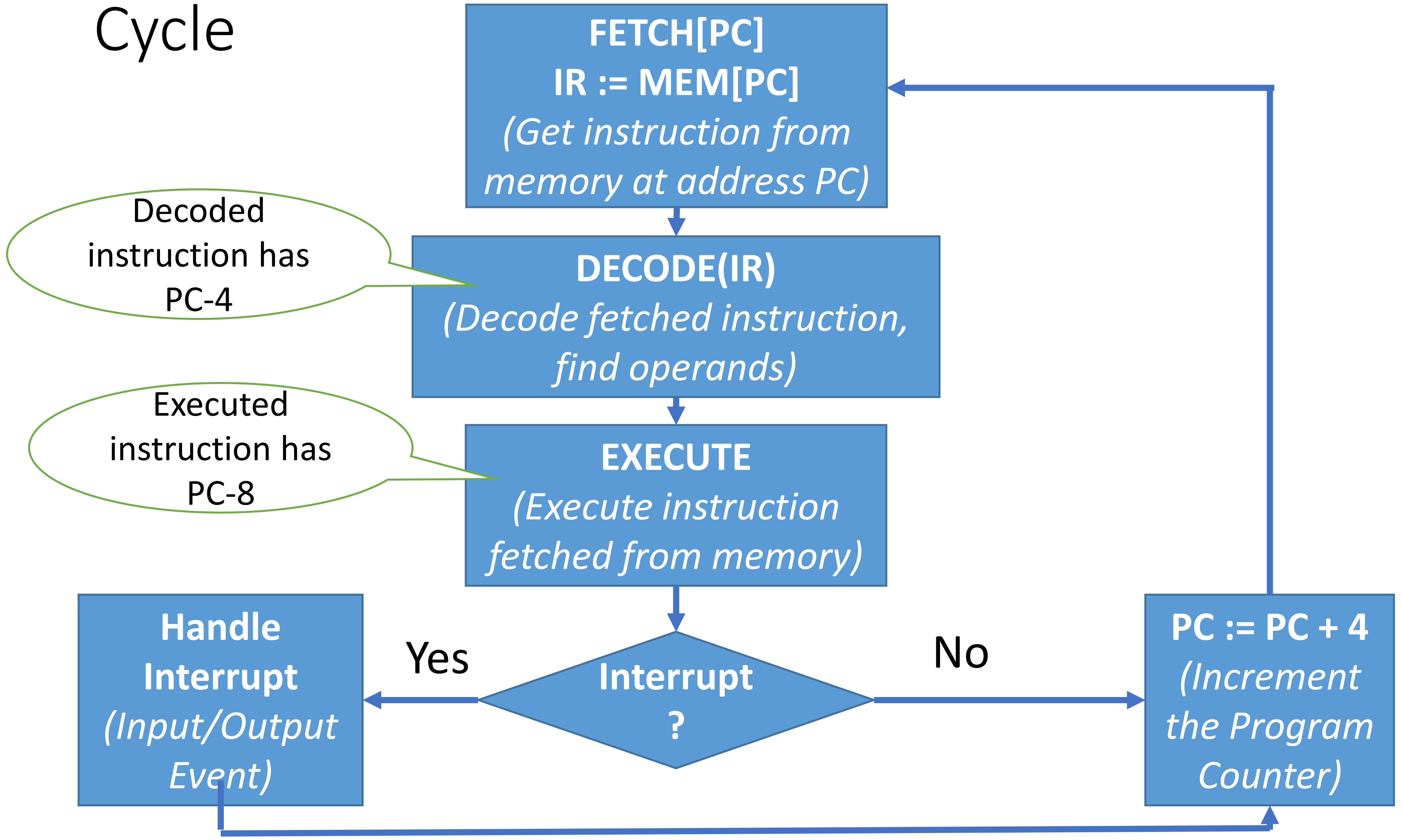
- Stages: fetch, decode, execute
- PC value = instruction being fetched
- PC – 4: instruction being decoded
- PC – 8: instruction being executed

- Beefier ARM variants use deeper pipelines (5 stages, 13 stages)

Recall: Abstract Processor Execution Cycle (*Simplified*)


FETCH[PC]
(Get instruction from memory)
EXECUTE
(Execute instruction fetched from memory)
Interrupt ?
Yes
No
Handle Interrupt
(Input/Output Event)
PC++
(Increment the Program Counter)

ARM 3-Stage Pipeline Processor Execution Cycle


FETCH[PC]
IR := MEM[PC]
(Get instruction from memory at address PC)
DECODE(IR)
(Decode fetched instruction, find operands)
EXECUTE
(Execute instruction fetched from memory)
Interrupt ?
Yes
No
Handle Interrupt (Input/Output Event)
PC := PC + 4 (Increment the Program Counter)
Decoded instruction has PC-4
Executed instruction has PC-8

String Output

- So far we have seen character input/output
- That is, one char at a time

- What about strings (character arrays, i.e., multiple characters)?

- Strings are stored in memory at consecutive addresses
  - Like arrays that we saw last time

```
string_abc:
.asciz "abcdefghijklmnopqrstuvwxyz\n\r"
.word 0x00
```

ADDR	Byte 3	Byte 2	Byte 1	Byte 0
0x1000	'd'	'c'	'b'	'a'
0x1004	'h'	'g'	'f'	'e'
0x1008	'l'	'k'	'j'	'i'
0x100c	'p'	'o'	'n'	'm'
0x1010	't'	's'	'r'	'q'
0x1014	'x'	'w'	'v'	'u'
0x1018	'\0'	'\0'	'z'	'y'

Assembler Output

```
0001012e <string_abc>:
   1012e: 64636261   strbtvs    r6, [r3], #-609; 0x261
   10132: 68676665   stmdavs    r7!, {r0, r2, r5, r6, r9, sl, sp,
lr}^
   10136: 6c6b6a69   stclvs     10, cr6, [fp], #-420; 0xfffffe5c
   1013a: 706f6e6d   rsbvc      r6, pc, sp, ror #28
   1013e: 74737271   ldrbtvc    r7, [r3], #-625; 0x271
   10142: 78777675   ldmdavc    r7!, {r0, r2, r4, r5, r6, r9, sl,
ip, sp, lr}^
   10146: 0d0a7a79   vstreq     s14, [sl, #-484]    ; 0xfffffe1c
   1014a: 00000000   andeq      r0, r0, r0
```

ASCII

Binary	Octal	Decimal	Hex	Glyph
110 0000	140	96	60	`
110 0001	141	97	61	a
110 0010	142	98	62	b
110 0011	143	99	63	c
110 0100	144	100	64	d
110 0101	145	101	65	e
110 0110	146	102	66	f
…				…
111 1000	170	120	78	x
111 1001	171	121	79	y
111 1010	172	122	7A	z

Printing Strings

```
@ assumes r0 contains uart data register address
@ r1 should contain address of first character of string
@ to display; stop if 0x00 (‘\0’) seen
print_string: push  {r1,r2,lr}
str_out: ldrb  r2,[r1]
     cmp  r2,#0x00  @ '\0' = 0x00: null character?
     beq  str_done  @ if yes, quit
     str  r2,[r0]   @ otherwise, write char of string
     add  r1,r1,#1  @ go to next character
     b    str_out   @ repeat
str_done: pop  {r1,r2,lr}
     bx   lr
```

Summary

- Chapter 1
- Chapter 2 (ARM)
- Relevant appendices for extra details

Questions?

?