ENGN2219/COMP6719
Computer Systems & Organization

Convener: Shoaib Akram

shoaib.akram@anu.edu.au

Australian National University

Plan: Lectures

- C and assembly
  - *Hardware/software interaction*
- Memory and storage devices
  - *How do the devices work?*
  - *How are they exposed to C programs?*
- Hardware optimizations
  - *Caches and virtual memory (memory-side)*
  - *Pipelinining (CPU-side)*

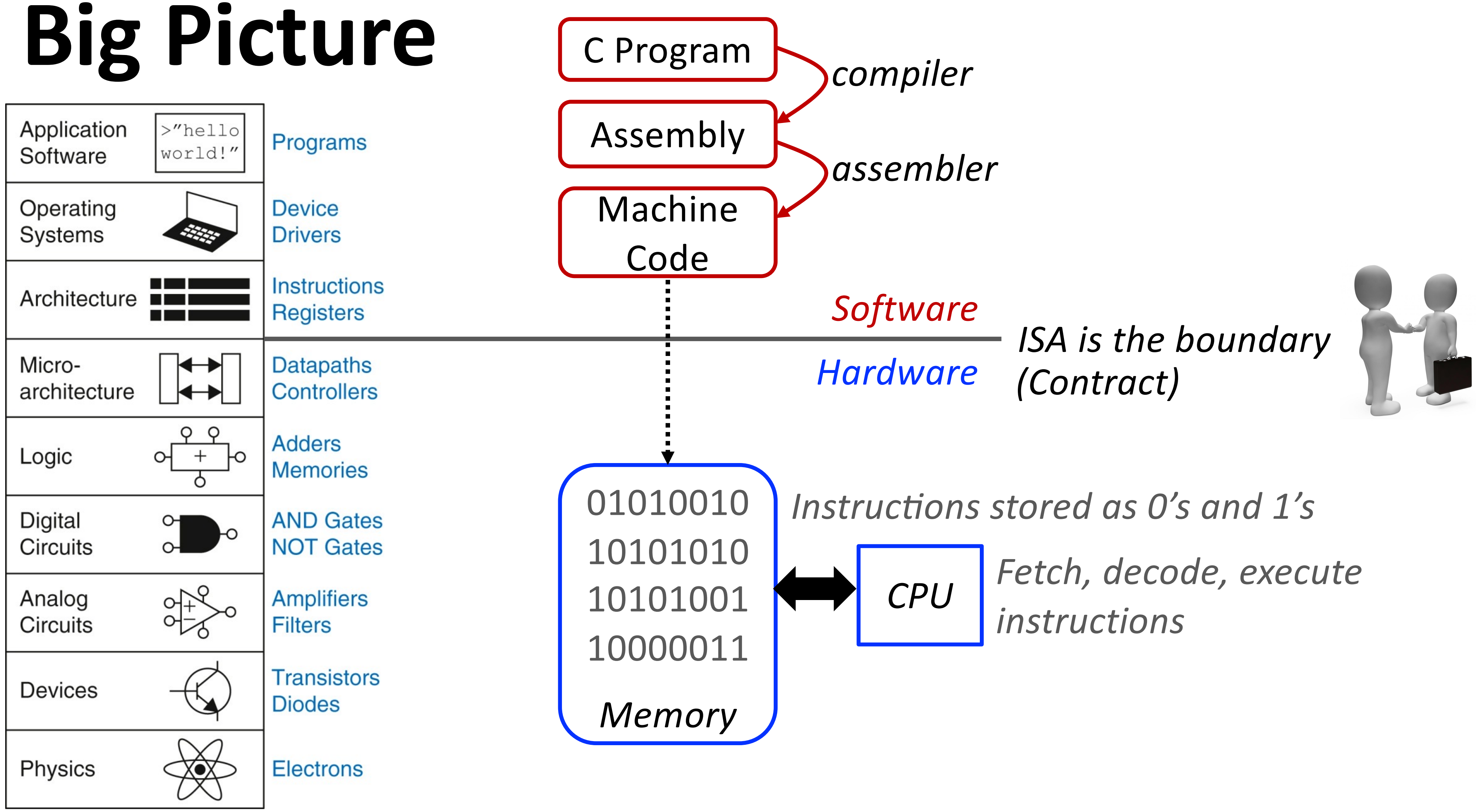
Plan: Labs

- Introduction to C – Part 1 ✓
- Introduction to C – Part 2
  - Control flow, bitwise operations, more pointers, strings
- Data Structures (beyond arrays)
  - Structs, unions, linked lists, read/write file I/O
- Dynamic Memory Allocation (rich topic)
- Assignment 2
  - Problem specification (QuAC CPU model, memory allocator)
  - Your task: Solve the problem in C (knowing assembly helps)
  - If you do Labs 1 – 3 diligently, you will (mostly) nail it!

Hardware/Software Interaction

- *Predominantly Assembly, some C (2 lectures)*
- *Exclusively C (2 – 3 lectures)*

Big Picture



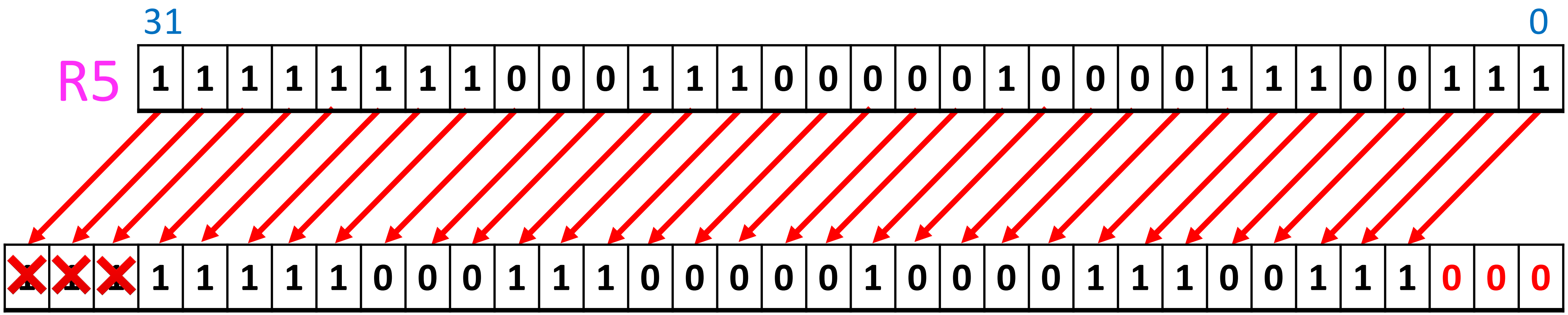
Application Software
>"hello world!"
Programs
Operating Systems
Device Drivers
Architecture
Instructions Registers
Micro-architecture
Datapaths Controllers
Logic
Adders Memories
Digital Circuits
AND Gates NOT Gates
Analog Circuits
Amplifiers Filters
Devices
Transistors Diodes
Physics
Electrons
C Program
compiler
Assembly
assembler
Machine Code
Software
Hardware
ISA is the boundary (Contract)
01010010
10101010
10101001
10000011
Memory
Instructions stored as 0's and 1's
CPU
Fetch, decode, execute instructions

Shift Instructions

- Shift the value in a register left or right, drop bits off the end
  - Logical shift left (LSL)
  - Logical shift right (LSR)
  - Arithmetic shift right (ASR)
  - Rotate right (ROR)
- Logical shift: shifts the number to the left or right and fills the empty slots with zero
- Arithmetic shift: on right shifts fill the most significant bits with the sign bit
- Rotate: rotates number in a circle such that empty spots are filled with bits shifted off the other end

Logical Shift Left (LSL)

ARM Instruction `LSL R0, R5, #3`



- Shift all bits left 3 positions, insert three 0's from the left
- Drop the 3 bits from the right

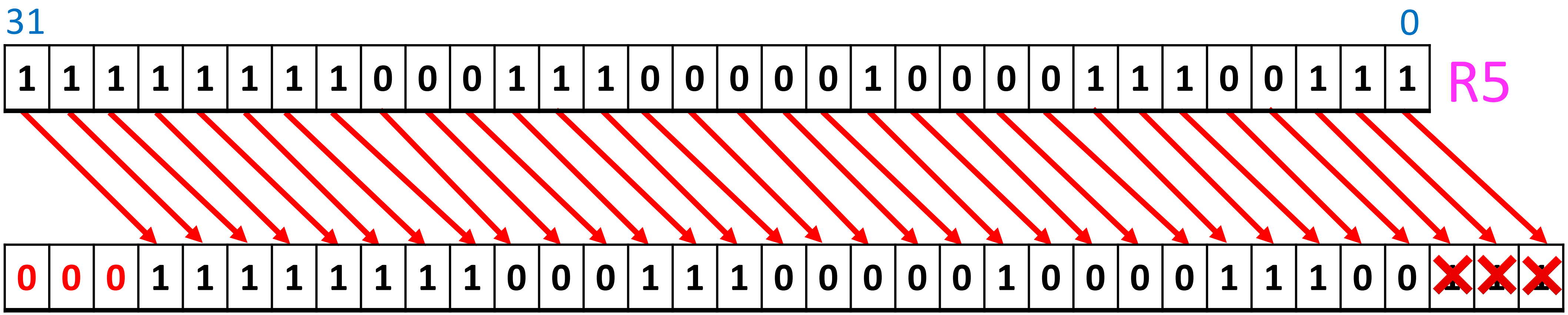
Result

R0	1	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	1	1	1	0	0	0

Logical Shift Right (LSR)

ARM Instruction

LSR	R0,	R5,	#3



- Shift all bits right 3 positions, insert three 0's from the right
- Drop the 3 bits from the left

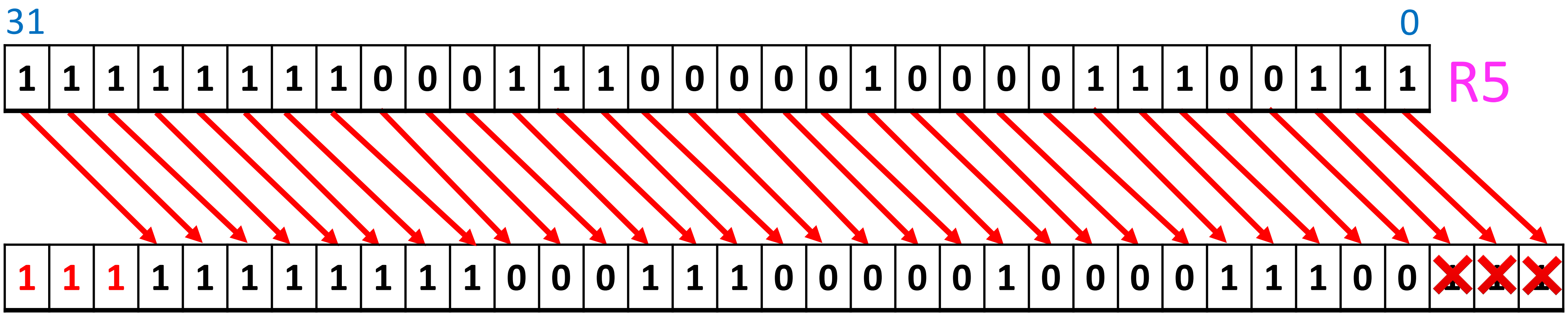
Result

0	0	0	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	R0

Arithmetic Shift Right (LSR)

ARM Instruction

ASR	R0,	R5,	#3



- Shift all bits right 3 positions, insert three 0's from the right
- Drop the 3 bits from the left

Result

1	1	1	1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	R0

Rotate Right (ROR)

ARM Instruction

ROR	R0,	R5,	#21

31											20																				0	
1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	1	1	1	R5

- Do a circular shift
- Right shift by 21 and put back bits that fall off at left end

31											20																				0	
1	1	1	1	1	1	1	1	0	0	0	1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	1	1	1	R5

Result

1	1	1	0	0	0	0	0	1	0	0	0	0	1	1	1	0	0	1	1	1	1	1	1	1	1	1	1	1	0	0	0	R0

Shifts: Machine Representation

DP-R

31:28	27:26	25	24:21	20	19:16	15:12	11:4								3:0
cond	00	0	cmd	S	Rn	Rd	0	0	0	0	0	0	0	0	Rm

Shift Instructions

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
cond	00	0	cmd	S	Rn	Rd	shamt5	sh	0	Rm

- cmd = 1101
- sh = 00 (LSL), 01 (LSR), 10 (ASR), 11 (ROR)
- Rn = 0
- shamt5 = 5-bit shift amount

Shifts: Machine Representation

- Format (Src2 = Register)

  LSL R0, R5, #3

  ↓ ↓ ↓

  LSL Rd, Rm, shamt5

31:28	27:26	25	24:21	20	19:16	15:12	11:7	6:5	4	3:0
cond	00	0	cmd	S	Rn	Rd	shamt5	sh	0	Rm

Shift Instructions

- **Immediate** shift amount (5-bit immediate)
- Shift amount: 0-31
- Rn is not used
- sh encodes the type of shift

- ARM also has instructions with shift amount held in a register

```
LSL   R4,   R8,   R6
```

```
ROR   R5,   R8,   R6
```

31:28	27:26	25	24:21	20	19:16	15:12	11:8	7	6:5	4	3:0
cond	00	0	cmd	S	Rn	Rd	Rs	0	sh	1	Rm

Shift Instructions

- Having dedicated instructions for shift operations is useful for systems programming
- Bit masks are a common requirement in low-level hardware resource management
- Code that manages network protocols or file formats
- Anything related to compression/decompression or packing/unpacking of information
- Rotation is used in cyclic codes (cryptography, compression)

Control Flow

- In real programs, the *order* in which statements execute is not always sequential (one after the other)
- Decisions and iterating the same task repeatedly is common
  - `if/else` ✓
  - `switch`
  - `for` and `while`
- How can we write these statements in assembly?
  - Performance analysis
  - Evaluate alternatives

For Loop in C: Sum

C code:

```
int i;
int sum = 0;

for (i = 0; i < 10; i = i + 1) {
    sum = sum + i;
}
```

C code:

```
int i;
int sum = 0;

for (i = 9; i >= 0; i = i – 1) {
    sum = sum + i;
}
```

Decremented version

- The variable “i” is called the loop index/counter
- i = 0 : index initialization
- i < 10 : loop termination condition
- i = i + 1 : loop advancement

Sum: Assembly

C code:

```c
int i;
int sum = 0;

for (i = 0; i < 10; i = i + 1) {
        sum = sum + i;
}
```

ARM Assembly code

; R0 = i, R1 = sum

```
        MOV     R1,     #0
        MOV     R0,     #0
FOR
        CMP     R0,     #10
        BGE     DONE
        ADD     R1,     R1,     R0
        ADD     R0,     R0,     #1
        B       FOR
DONE
```

check termination condition to break out of the loop if condition is met

keep iterating by branching back

Sum: Perf Analysis

ARM Assembly code

; R0 = i, R1 = sum

```
        MOV   R1,   #0
        MOV   R0,   #0
FOR
        CMP   R0,   #10
        BGE   DONE
        ADD   R1,   R1,   R0
        ADD   R0,   R0,   #1
        B     FOR
DONE
```

How long does it take to execute the loop (*frequency = 1 GHz, CPI = 1*)

- # instructions = ?
- Clock cycle time, T_c = ?
- Execution time = ?

Sum: Alternative Approach

```
        MOV   R1,   #0
        MOV   R0,   #0
COND
        CMP   R0,   #10
        BLT   FOR
        B     DONE
FOR
        ADD   R1,   R1,   R0
        ADD   R0,   R0,   #1
        B     COND
DONE
```

- More faithfully follow the for loop semantics in C
- Use BLT instead of BGE

How long does it take to execute the loop (*frequency = 1 GHz, CPI = 1*)

- Instruction count = ?
- Clock cycle time, T_c = ?
- Execution time = ?

Sum: Decremented Version

C code:

```
int i;
int sum = 0;

for (i = 9; i != 0; i = i - 1) {
    sum = sum + i;
}
```

ARM Assembly code

; R0 = i, R1 = sum

```
        MOV   R1,  #0
        MOV   R0,  #9
FOR
        ADD   R1,  R1,  R0
        SUBS  R0,  R0,  #1
        BNE   FOR
DONE
```

sum = sum + 1

i = i – 1 and set flags

Saves 2 instructions per iteration:

- Decrement loop variable & compare: `SUBS R0, R0, #1`
- Only 1 branch – instead of 2

Exercise

- Find the time it takes to execute the loop now if the clock cycle time is 1 ns

Lessons

Execution time depends on how we write code and microarchitecture details

Always make the common case fast!

Note: *Eliminating a branch is always desirable in pipelined processors because the CPU needs to wait for the branch to finish execution in order to fetch the next instruction*

While Loop in C

- For loop iterate N times
  - Used when N is known in advance
- While loop
  - Iterate until the *controlling condition* is false
- Determine x such that $2^x = 128$

C code:

```
int pow = 1;
int x = 0;

while (pow != 128) {
    pow = pow * 2;
    x = x + 1;
}
```

Two Interesting While Loops

C code:

```
while (1) {
    // iterates forever
}

while (0) {
    // iterates 0 times
}
```

While Loop: C and Assembly

C code:

```c
int pow = 1;
int x = 0;

while (pow != 128) {
	pow = pow * 2;
	x = x + 1;
}
```

Determine x such that

$2^x = 128$

ARM Assembly code

; R0 = pow, R1 = x

```
        MOV   R0,   #1
        MOV   R1,   #0
WHILE
        CMP   R0,   #128
        BEQ   DONE
        LSL   R0,   R0,   #1
        ADD   R1,   R1,   #1
        B     WHILE
DONE
```

Exercise

- Find the time it takes to execute the While loop if the clock cycle time is 1 ns.

- Write `sum` as a while loop and find the time it takes to execute the resulting loop if all CPU parameters are the same as before.

switch/case Statement

C code:

```
switch(button) {
    case 1: atm = 20; break;
    case 2: atm = 50; break;
    case 3: atm = 100; break;
    default; atm = 0;
}
```

C code for if...else ladder:

```
if (button == 1)
    atm = 20;
else if (button == 2)
    atm = 50;
else if (button == 3)
    atm = 100;
else
    atm = 0;
```

- Execute one of several blocks of code depending on the condition and *break* out of the entire switch block
- If no conditions are met, the *default* block is executed

switch/case Statement

C code:

```
switch(button) {
    case 1: atm = 20; break;
    case 2: atm = 50; break;
    case 3: atm = 100; break;
    default; atm = 0;
}
```

ARM Assembly code

```
; R0 = button, R1 = atm
        CMP     R0,     #1
        MOVEQ   R1,     #20
        BEQ     DONE
        CMP     R0,     #2
        MOVEQ   R1,     #50
        BEQ     DONE
        CMP     R0,     #3
        MOVEQ   R1,     #50
        BEQ     DONE
        MOV     R1,     #0
DONE
```

Arrays

- Arrays contain a collection of similarly typed elements
- Elements are stored contiguously in memory

int is 4 bytes on most architectures

C code:

```
int marks[5] = {19, 10, 8, 17, 9};
int a = marks[0];
int b = 2;
marks[3] = b;
```

Address	Data	Index	Element
⋮	⋮	⋮	⋮
00000010	9	4	marks[4]
0000000C	17	3	marks[3]
00000008	8	2	marks[2]
00000004	10	1	marks[1]
00000000	19	0	marks[0]

4 Bytes

Array of Characters

- Array of **char**acters (**char** is a data type in C)
- **char** is used for representing characters

char is always 1 byte

C code:

```
char alphas[5] = {'a', 'b', 'c', 'd', 'e'};
```

Address	Data	Index	Element
⋮	⋮	⋮	⋮
0000004	'e'	4	alphas[4]
00000003	'd'	3	alphas[3]
00000002	'c'	2	alphas[2]
00000001	'b'	1	alphas[1]
00000000	'a'	0	alphas[0]

1 Byte

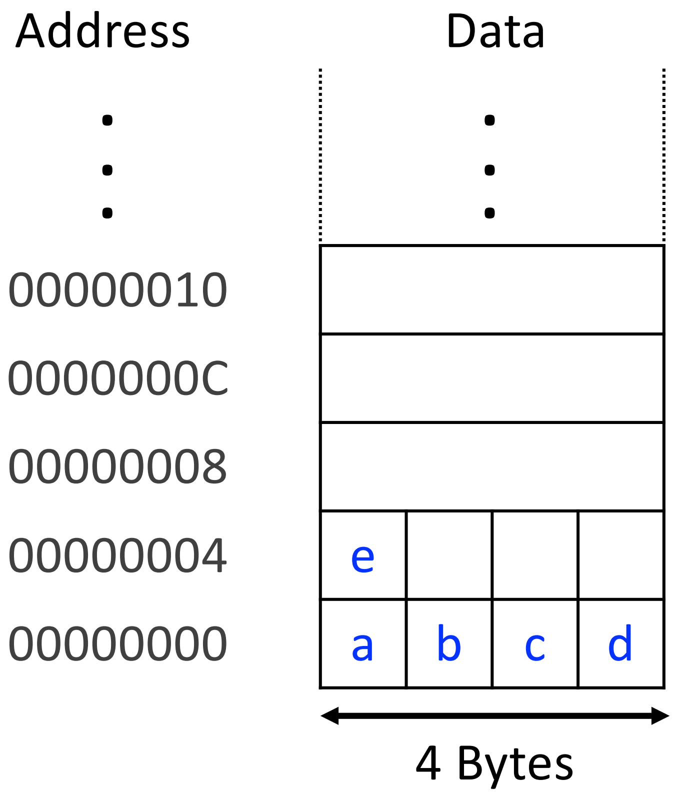
Array of Characters

- Array of **char**acters (char is a data type in C)
- char is used for representing characters

char is always 1 byte

C code:

```
char alphas[5] = {'a', 'b', 'c', 'd', 'e'};
```



Endianness

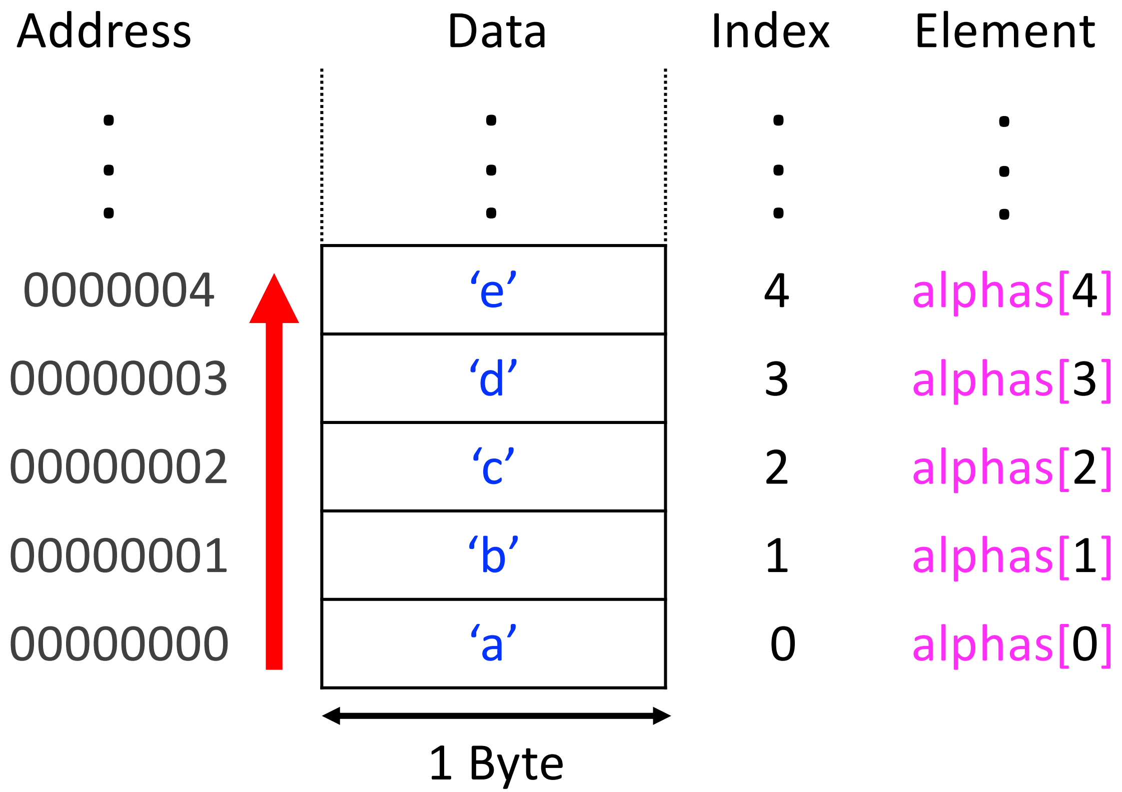
For large objects (greater than 1 byte), byte order in memory matters: *Which byte of a 4-byte int is stored at the lowest address?*

- Little Endian: Little end (LSB) stored first (at lowest address)
  - Intel x86
- Big Endian: Big end (MSB) stored first,
  - SPARC, Motorola processors
- ARM is bi-endian (supports both)

Characters and Endianness

- Characters are 1-byte each
- There is no ambiguity in which byte to store first
  - Always Big-Endian

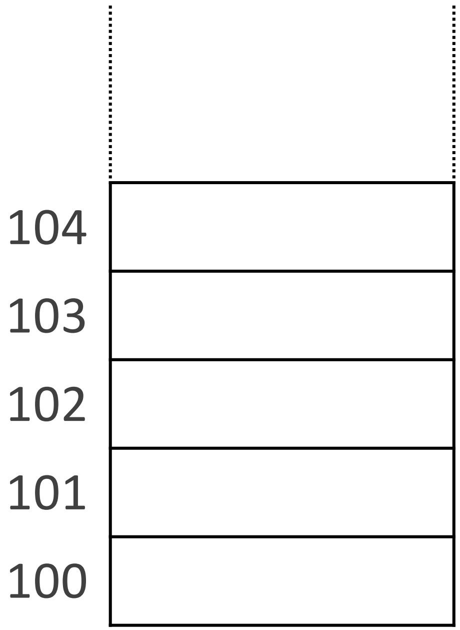
```
char alphas[5] = {'a', 'b', 'c', 'd', 'e'};
```



alphas[0] is stored in memory at address 0x00000000

Integers and Endianness

- A 32-bit integer is stored at memory address 100
- Stored in locations: 100, 101, 102, 103
- Which part of the 32-bit value is stored first?

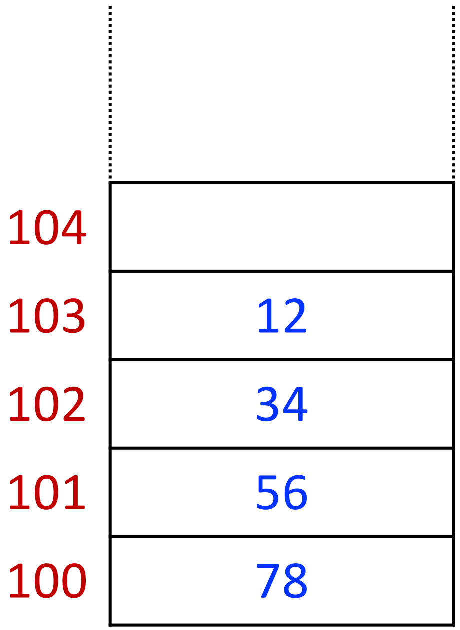


00010010001101000101011001111000

Hex 12 34 56 78

Integers and Endianness

- A 32-bit integer is stored at memory address 100
- Stored in locations: 100, 101, 102, 103
- Which part of the 32-bit value is stored first?



00010010001101000101011001111000

Hex 12 34 56 78

Little Endian: 78 at location 100

Integers and Endianness

- A 32-bit integer is stored at memory address 100
- Stored in locations: 100, 101, 102, 103
- Which part of the 32-bit value is stored first?

Address	Data
104	
103	78
102	56
101	34
100	12

00010010001101000101011001111000

Hex 12 34 56 78

Big Endian: 12 at location 100

Endianness: Pros

- Little Endian
  - Easy to create small values from large values
  - Previous example: Read byte at address 100
  - On Big Endian, add 4 to 100, then read byte to find out
- Big Endian
  - Easy to test sign and range of a value

QuAC architecture in labs evades the entire issue of endianness with word-addressable memory

Array Sum

- Example to illustrate how instructions are picked for ISAs
- Add a constant 10 to each element of the scores array

C code:

```c
int i;
int scores[200];
// initialization code
...
for (i = 0; i<200; i++)
        scores[i] = scores[i] + 10;
```

Assembly code:

```
; R0 = array base address, R1 = i
MOV    R0,  #0x14000000      ; R0 = base address
MOV    R1,  0                ; i = 0
LOOP
  CMP  R1,  #200             ; i < 200?
  BGE  L3                    ; if not, exit loop
  LSL  R2,  R1,  #2          ; R2 = i*4
  LDR  R3,  [R0, R2]         ; R3 = scores[i]
  ADD  R3,  R3,  #10         ; R3 = scores[i] + 10
  STR  R3,  [R0, R2]         ; scores[i] += 10
  ADD  R1,  R1,  #1          ; i = i + 1
  B    LOOP                  ; repeat loop
L3
```

LDR with register as offset

- New LDR variant (LDR with register as offset)

```
LDR   R3,   [R0, R2]
```

dest   base   offset

- Very common to load from memory with base + offset addressing, so there is an instruction for that
- R2 is called the index register

Condensing Array Sum – 1

- LSL/LDR combo often used in tandem in array traversals
  - There is support for that in the ISA
- Eliminates LSL instruction

ARM has an instruction that scales index reg. R1

LDR R3, [R0, R1, LSL #2]

Left shift is a multiply by 2

Memory address = R0 + (R1 * 4)

```
Assembly code:
; R0 = array base address, R1 = i
MOV   R0,  #0x14000000
MOV   R1,  0
LOOP
  CMP   R1,  #200
  BGE   L3
  LSL   R2,  R1,  #2
  LDR   R3,  [R0, R2]
  ADD   R3,  R3,  #10
  STR   R3,  [R0, R2]
  ADD   R1,  R1,  #1
  B     LOOP
L3
```

- R0 = base address
- i = 0
- i < 200?
- if not, exit loop
- R2 = i*4
- R3 = scores[i]
- R3 = scores[i] + 10
- scores[i] += 10
- i = i + 1
- repeat loop