Version 1.00

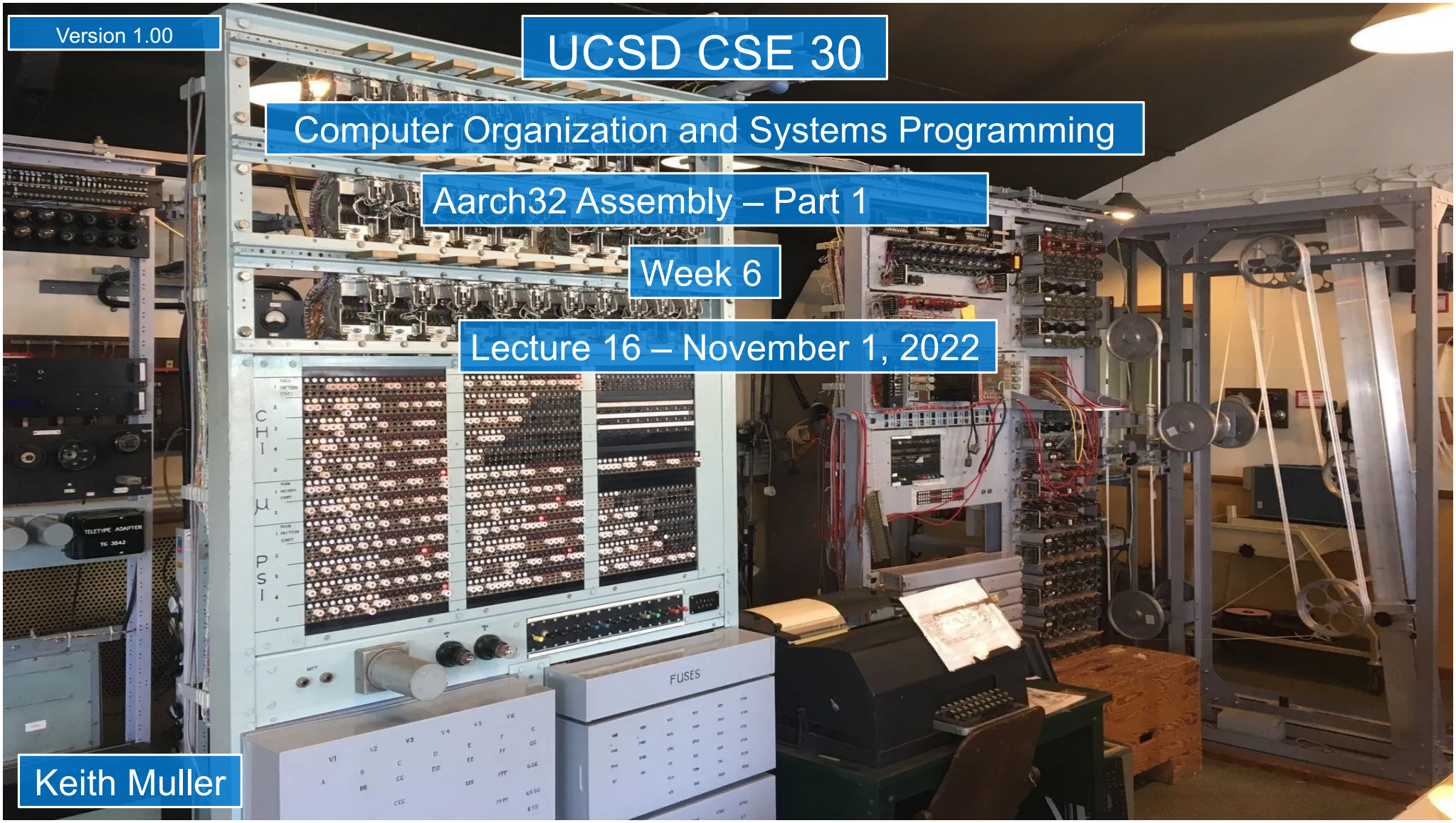
UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Part 1

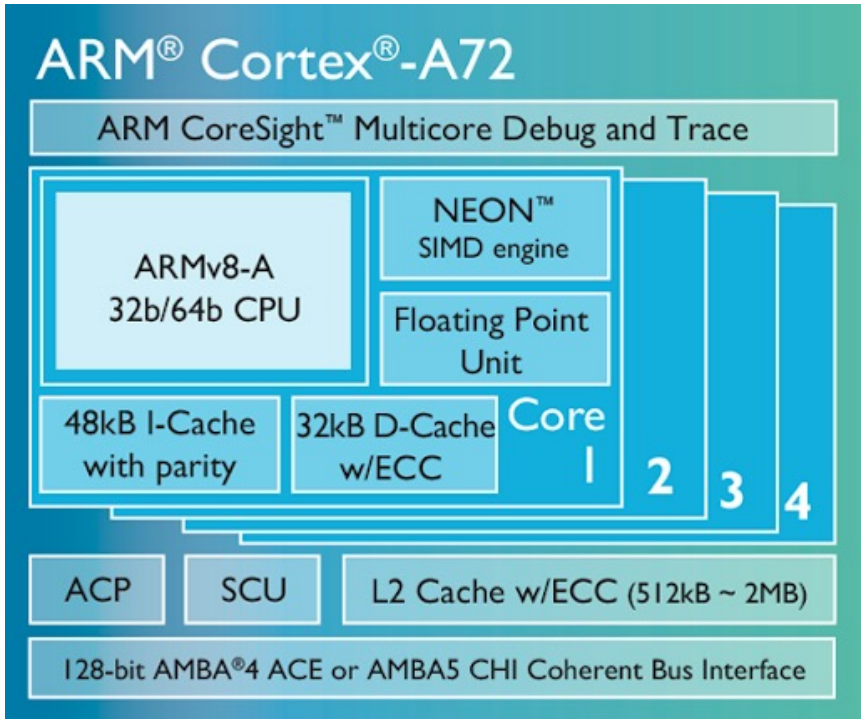
Week 6

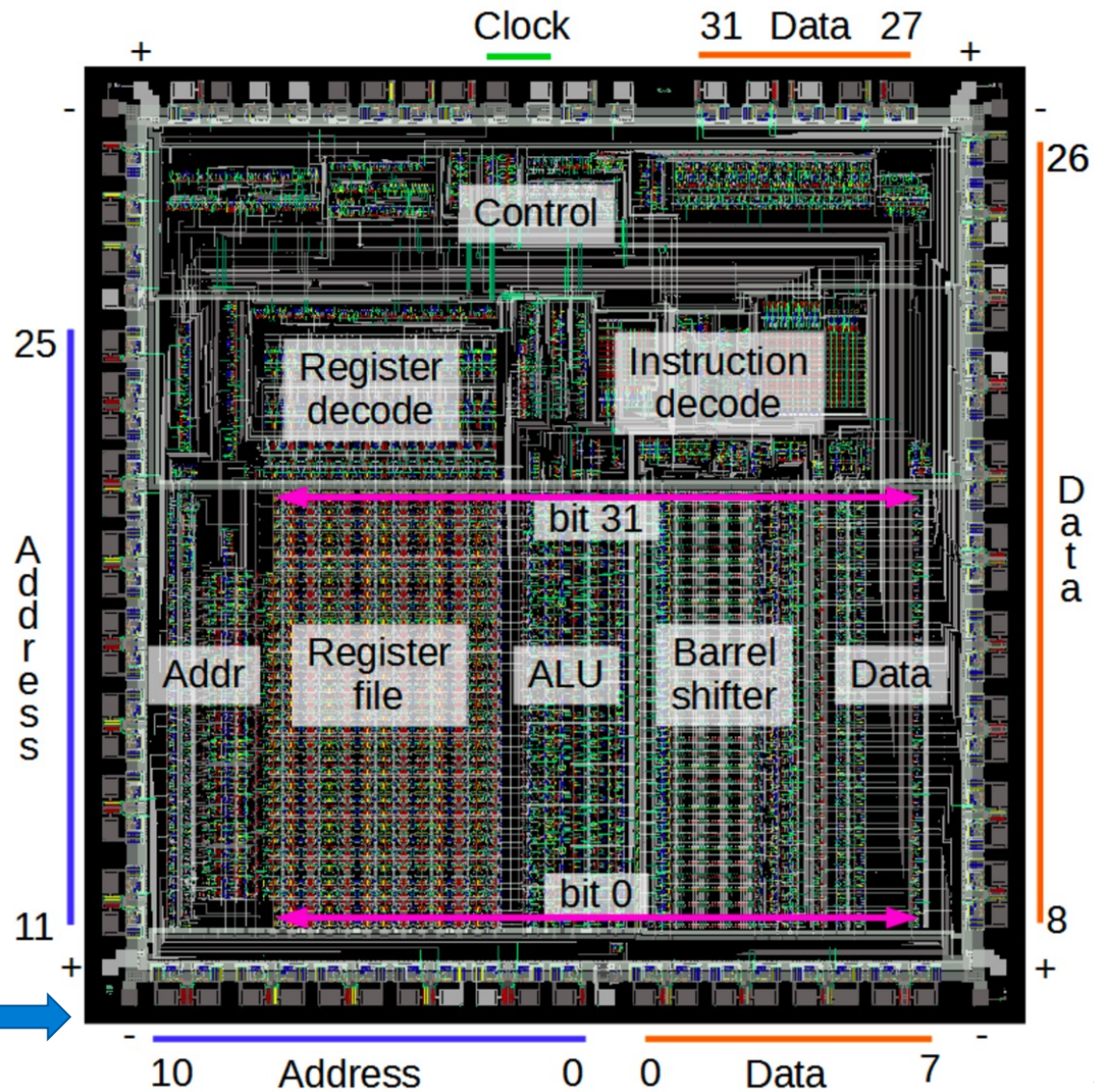
Lecture 16 – November 1, 2022



Keith Muller

Arm Core Organization & Floorplan Examples





Single core ***arm*** die (not an A72) ***Floorplan***

x

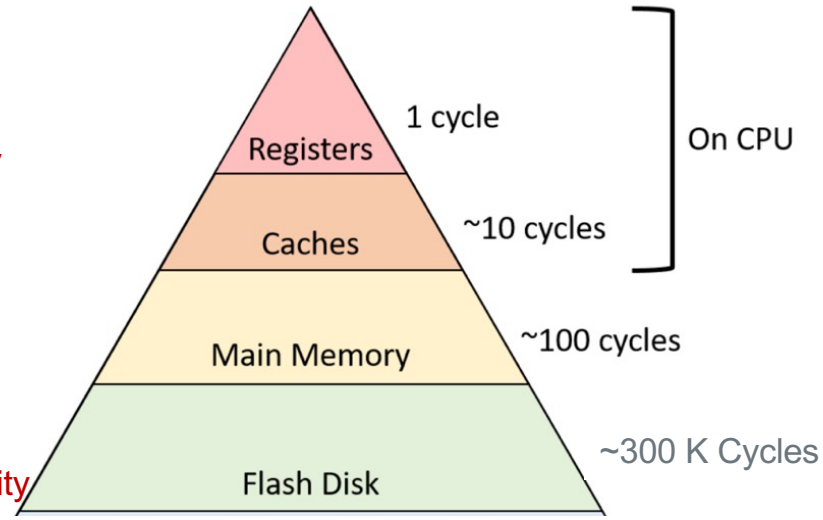
Memory Triangle: Hardware Cost/Performance/Capacity Tiers

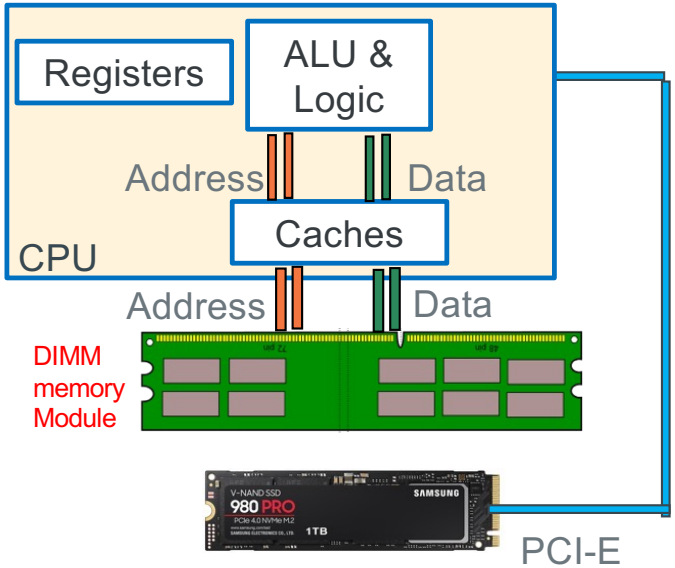
Assume 1 clock cycle per machine instruction

Goal: keep most Data Accesses high in the Triangle

1. Smallest Capacity
2. Highest Performance
3. Highest cost/capacity

1. Largest Capacity
2. Slowest performance
3. Lowest Cost $/capacity





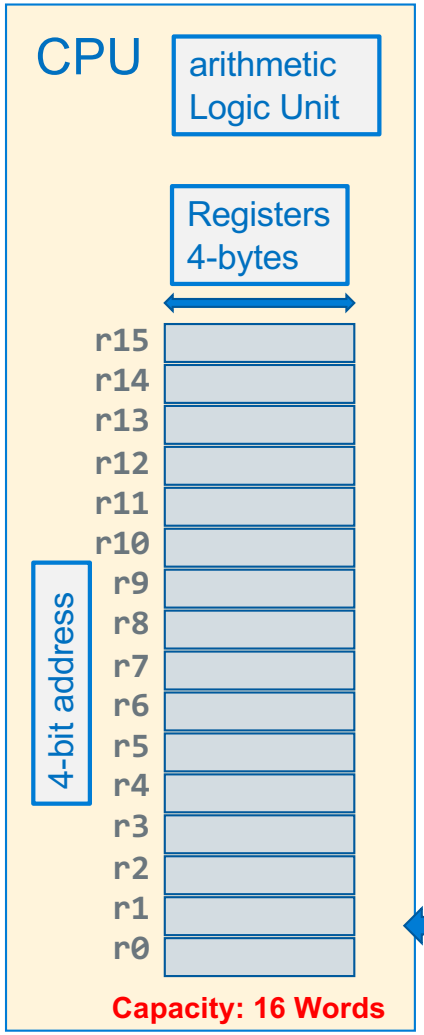
Clock cycle =~ time to access; larger is slower

Design Tradeoff: Based on workload considering cost and performance targets

Source: Dive Into Systems Text

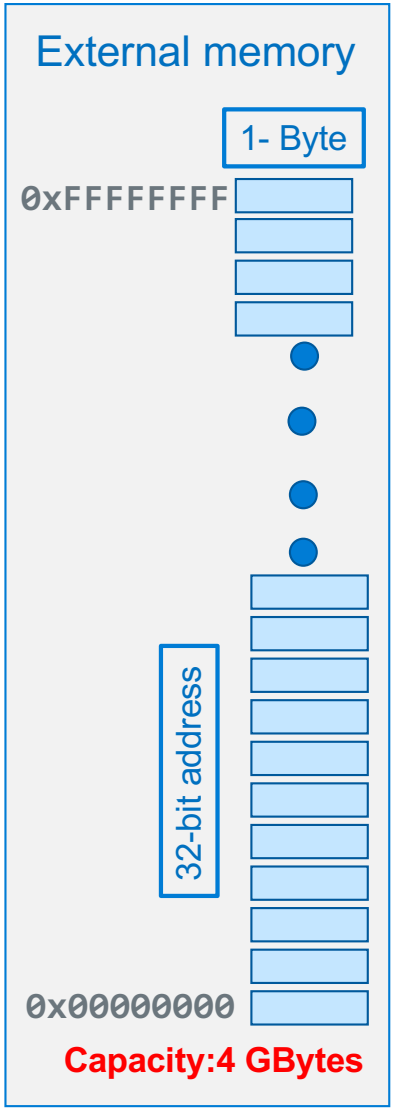
x

32-Bit Arm - Registers



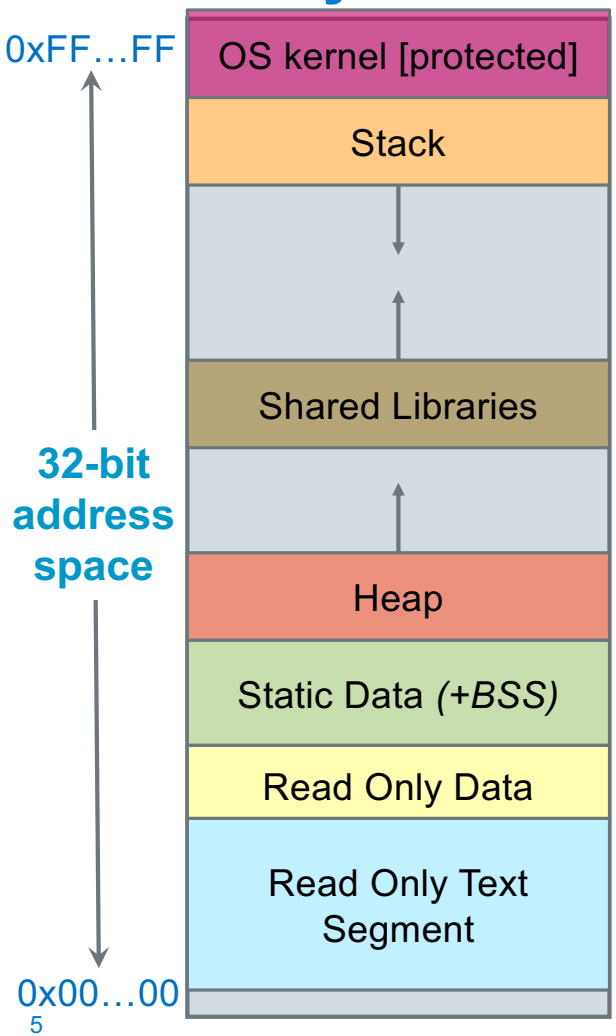
- Registers are **memory located within the CPU**
- **Registers are the <u>fastest</u>** read and write storage
- Register is **word size in length** stores 32-bit values
  - Memory is accessed using pointers in registers
- In assembly language the registers have predefined names starting with an r to differentiate them from memory addresses which are labels (address)
- 16 registers: from r0 to r15 (encoded: 0x0 – 0xf)

CPU Memory Bus = Address + Data



x

Assembly and Machine Code



- Machine Language (or code): Set of instructions the CPU executes are encoded in memory using patterns of ones and zeros (like binary numbers)
- Assembly language is a symbolic version of the machine language
- Each assembly statement (called an **Instruction**)
  - Executes exactly **one** from a list of simple commands
  - Instructions describe operations (e.g., =, +, -, *)
  - Execution flows from low to high memory one instruction at a time unless there is a branch
- One line of arm32 machine code contains one instruction in one word (32 bits)
- Assembler (gnu as) translates assembly to machine code

Memory Address	word (4-bytes) contents	Assembly Language
1040c:	e28db004	add fp, sp, 4
10410:	e59f0010	ldr r0, [pc, 16]
10414:	ebffffb3	bl 102e8 <printf>
10418:	e3a00000	mov r0, 0
1041c:	e24bd004	sub sp, fp, 4

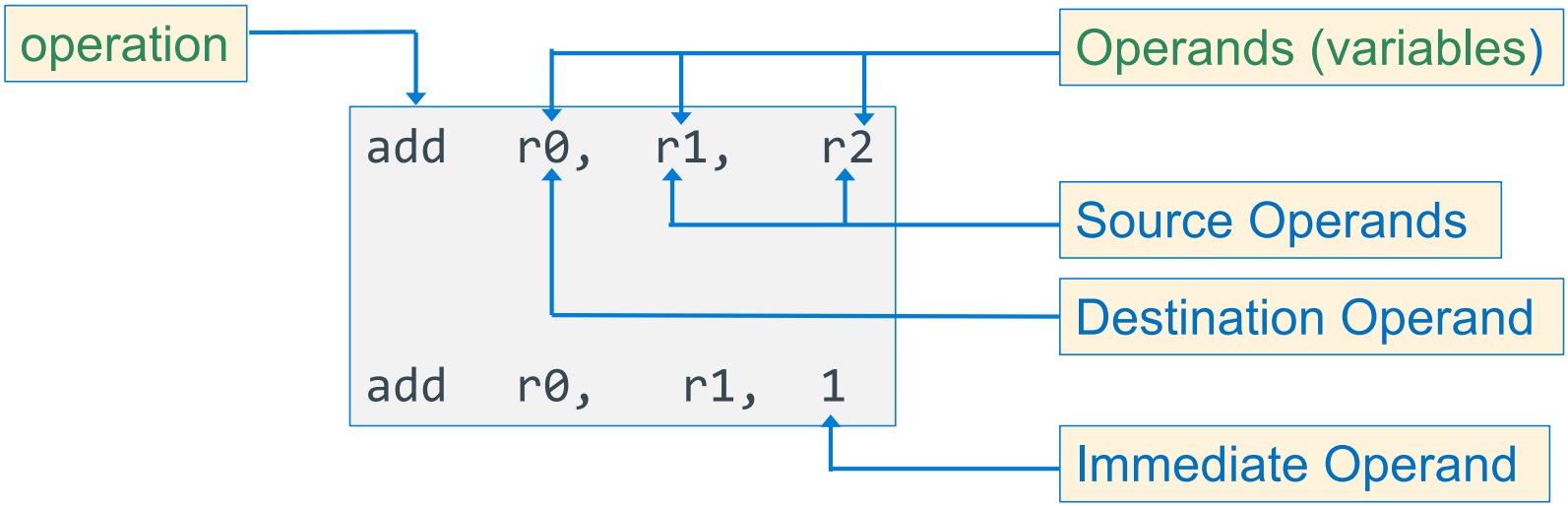
Machine Code

high <- low bytes

x

Anatomy of an Assembly instruction

- Assembly language instructions specify an operation and the operands to the instruction (arguments of the operation)
- Three basic types of operands
  - Destination: where the result will be stored
  - Source: where data is read from
  - Immediate: an actual value like the 1 in y = x + 1



x

Meaning of an Instruction

- Operations are abbreviated int opcodes (1– 5 letters)
- Assembly Instructions are specified with a very regular syntax
  - Opcodes are followed by arguments
  - Usually the destination argument is next, then one or more source arguments (this is not strictly the case, but it is generally true)
- Why this order?
- Analogy to C or Java

```
int r0, r1, r2;
    r0 = r1 + r2;          // c
```

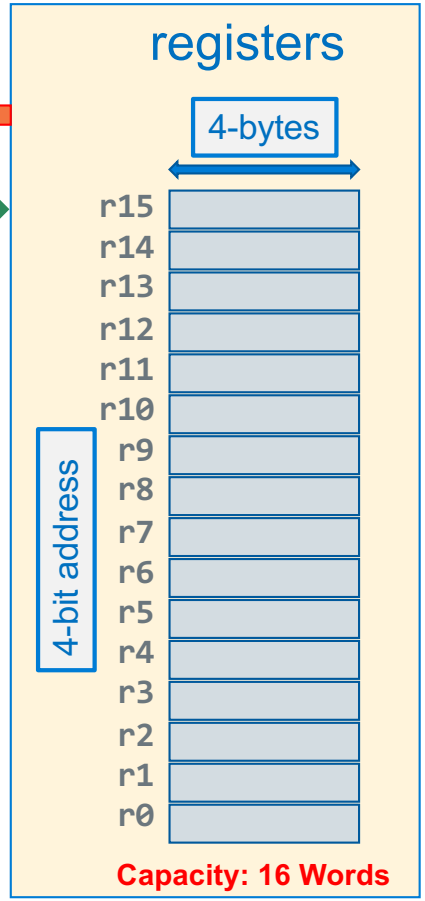
```
            r0 = r1  + r2
    add     r0,  r1,   r2   // assembly
```


x

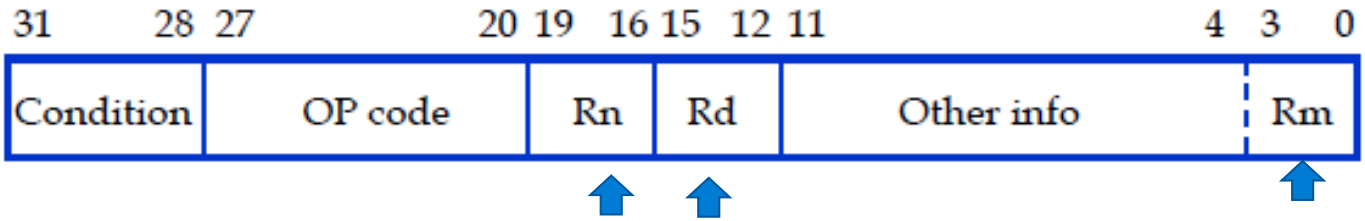
32-Bit Arm - Registers

All computations (add, subtract, etc.) are performed in the ALU

Arithmetic & Logic Unit (ALU)



- Almost all arithmetic, logic operations and data movement operations involve at least one register
- As a result, Register addresses are **directly** encoded into 4-bit fields in machine instructions (see below)

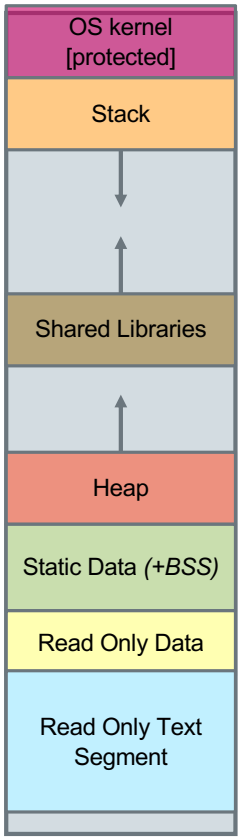


X

Program Execution: A Series of Instructions

- Instructions are **retrieved sequentially** from memory
- Each instruction executes to completion **before** the next instruction is completed
- Conceptually the pc (program counter) points at executing instruction
- exceptions: loops, function calls, traps,..

Main Memory

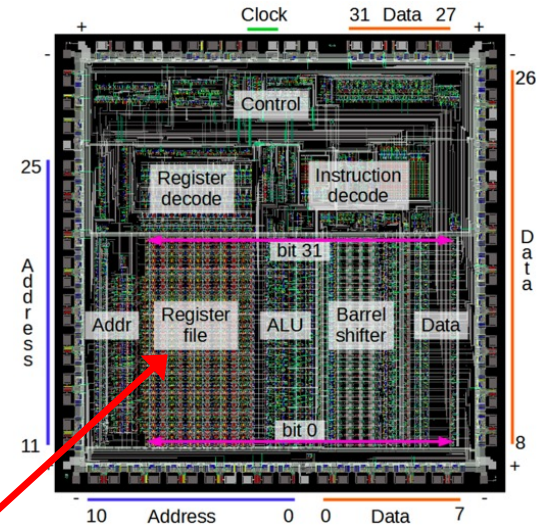


Memory Content in Text segment

```
add r0, r1, r1
add r0, r0, r0
add r0, r0, r0
sub r1, r0, r1
```

Low memory

Low memory



Register contents inside the CPU

initial values

```
r0 = 1 r1 = 2
```

```
r0 = 4 r1 = 2
```

```
r0 = 8 r1 = 2
```

```
r0 = 16 r1 = 2
```

```
r0 = 16 r1 = 14
```

Program Execution: Looping in the Execution Flow

- Repeat the series of instructions in a loop means altering the flow of execution
- This is used with if statements and loops
- Below is an infinite loop (br instruction: unconditional branch: "goto"

the address of this instruction has the name .Lloop

```
.Lloop:    add r0, r1, r1
           add r0, r0, r0
           add r0, r0, r0
           sub r1, r0, r1
           br .Lloop
```

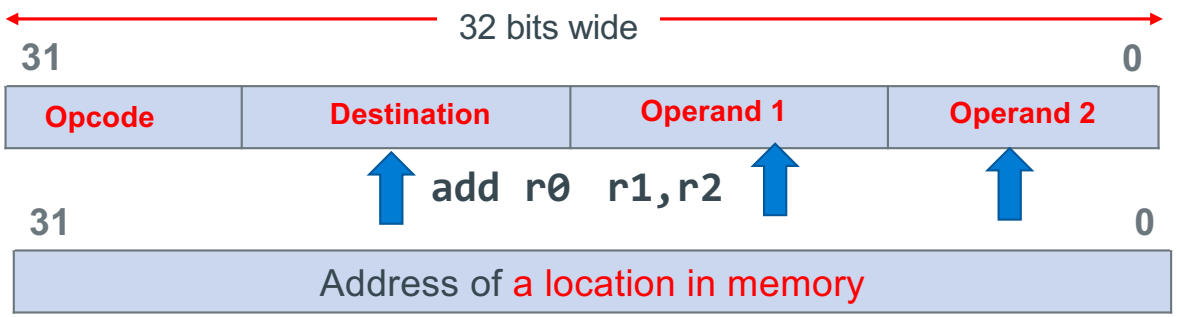
branch to the instruction at memory location with the label: .Lloop

```
r0 = 1 r1 = 2
r0 = 4 r1 = 2
r0 = 8 r1 = 2
r0 = 16 r1 = 2
r0 = 16 r1 = 14
```

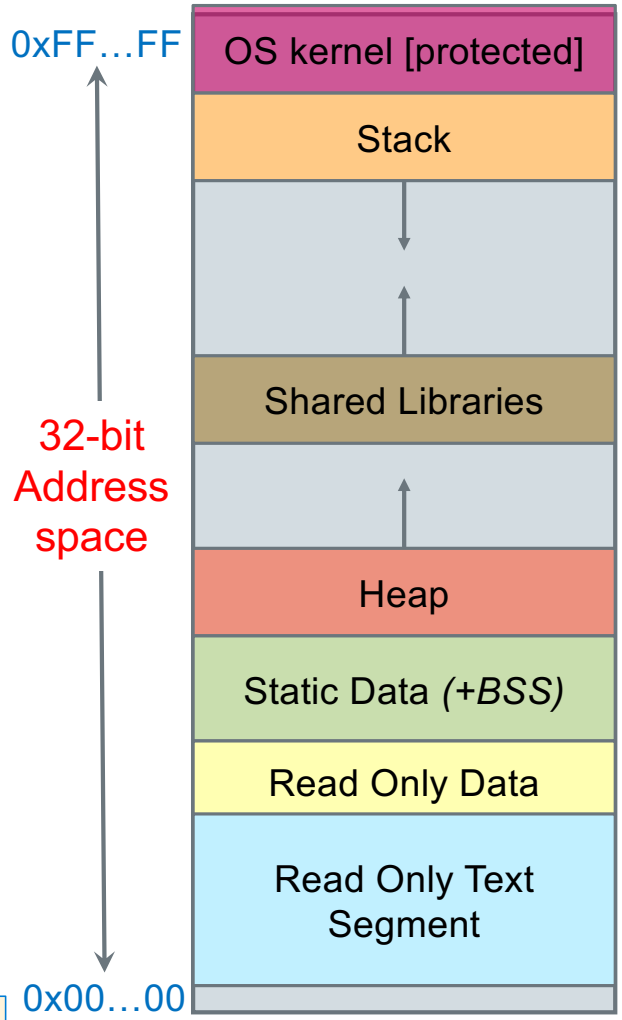

x

How to Access Memory?

- Consider r0 = r1 + r2
  - Operation code: add
  - Destination: r0
  - Operand 1: r1
  - Operand 2: r2
- Aarch32 Instructions are always word size: 32 bits wide
  - Some bits must be used to specify the operation code
  - Some bits must be used to specify the destination
  - Some bits must be used to specify the operands
- Address space is 32 bits wide – POINTERS in registers



NOT ENOUGH BITS for FULL Addresses in the instruction



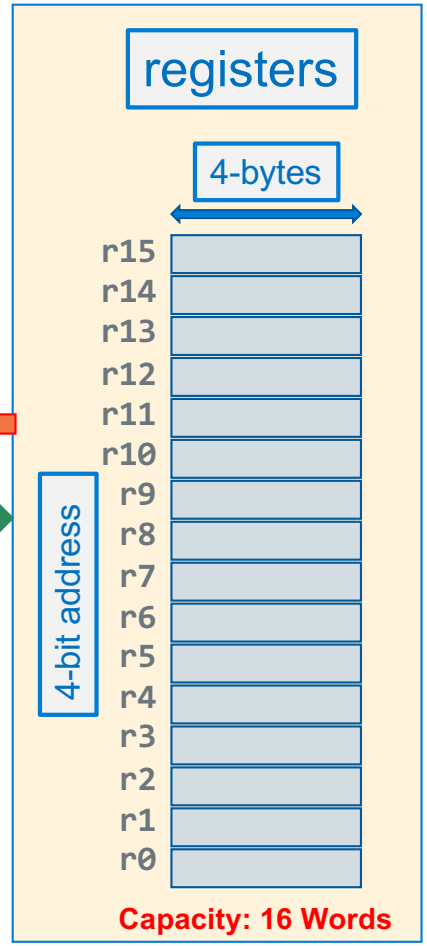
x

32-Bit Arm is a Load/Store Architecture

All computations (add, subtract, etc.) are performed in the ALU

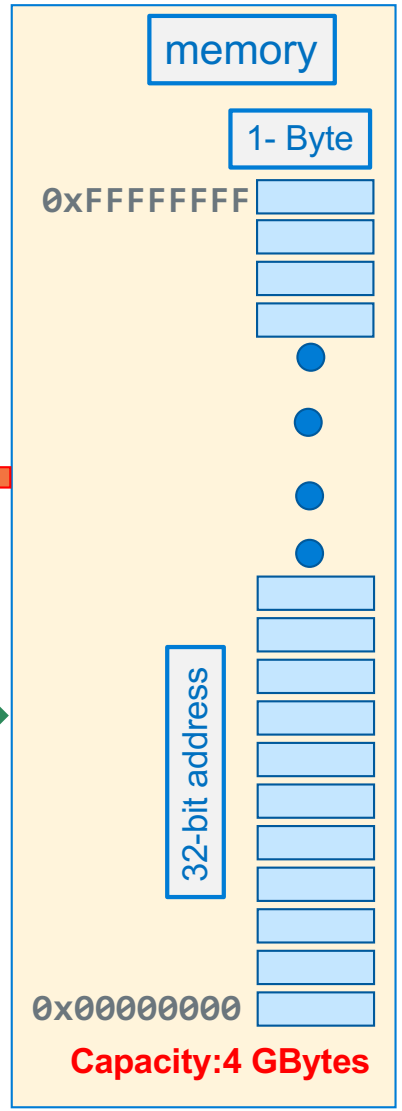
Arithmetic & Logic Unit (ALU)

Instruction operands and results are **only in registers**



Load:
(also called a Fill)
Copy the contents of a variable from memory into a register.

Store:
(also called a Spill)
Copy the contents of a register back into a variable in memory.



X

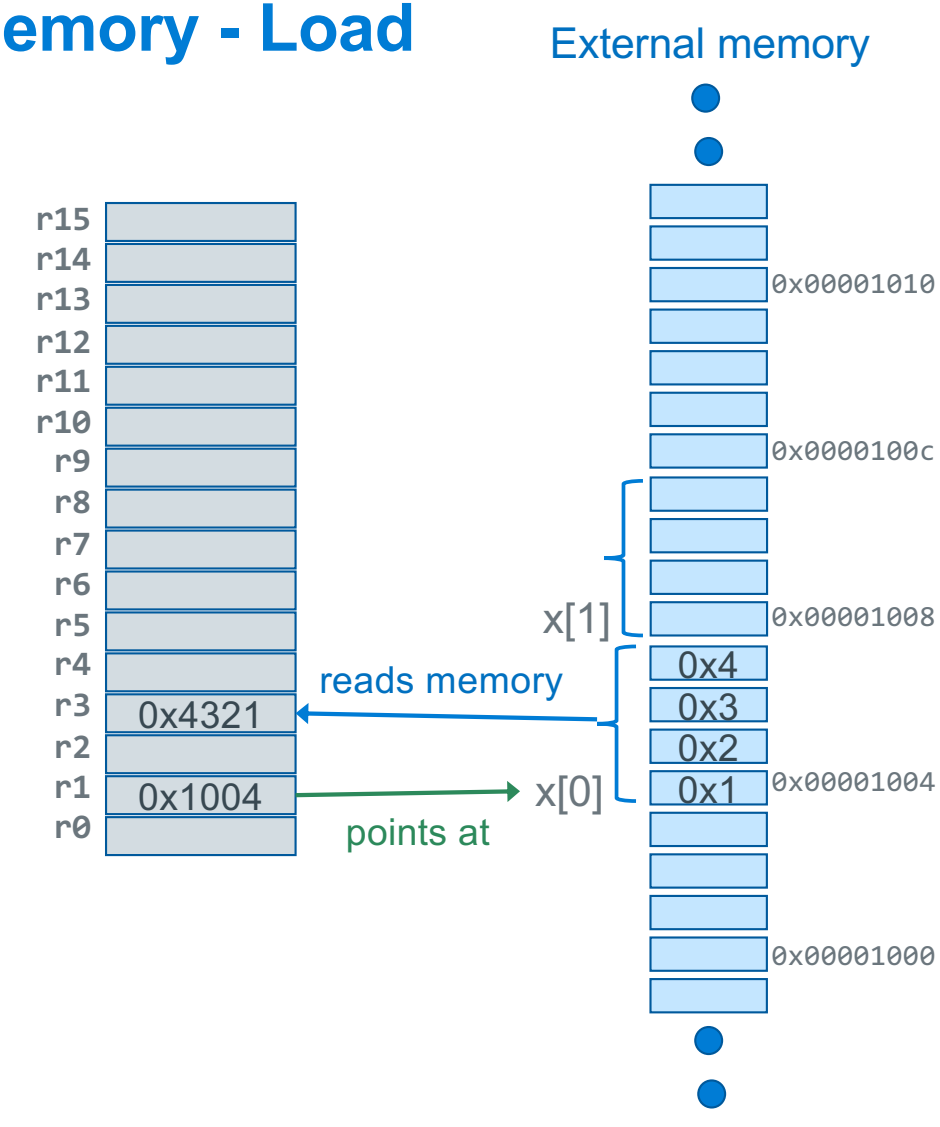
Using Registers as Pointers to Memory - Load

```
We want to do a x[1] = x[0]
We have to do this in two steps

int r3;
int x[2];
int *r1 = &x;     // r1 contains address
…
r3 = *(r1);       // memory to register
*(r1 + 1) = r3;   // register to memory
```

```
Load register from memory
ldr    r3, [r1, 0]
                ↑
address = r1 + 0 = 0x1004
```

The [ ] around the operands is like the * dereference op

we will cover this instruction in more detail later



x

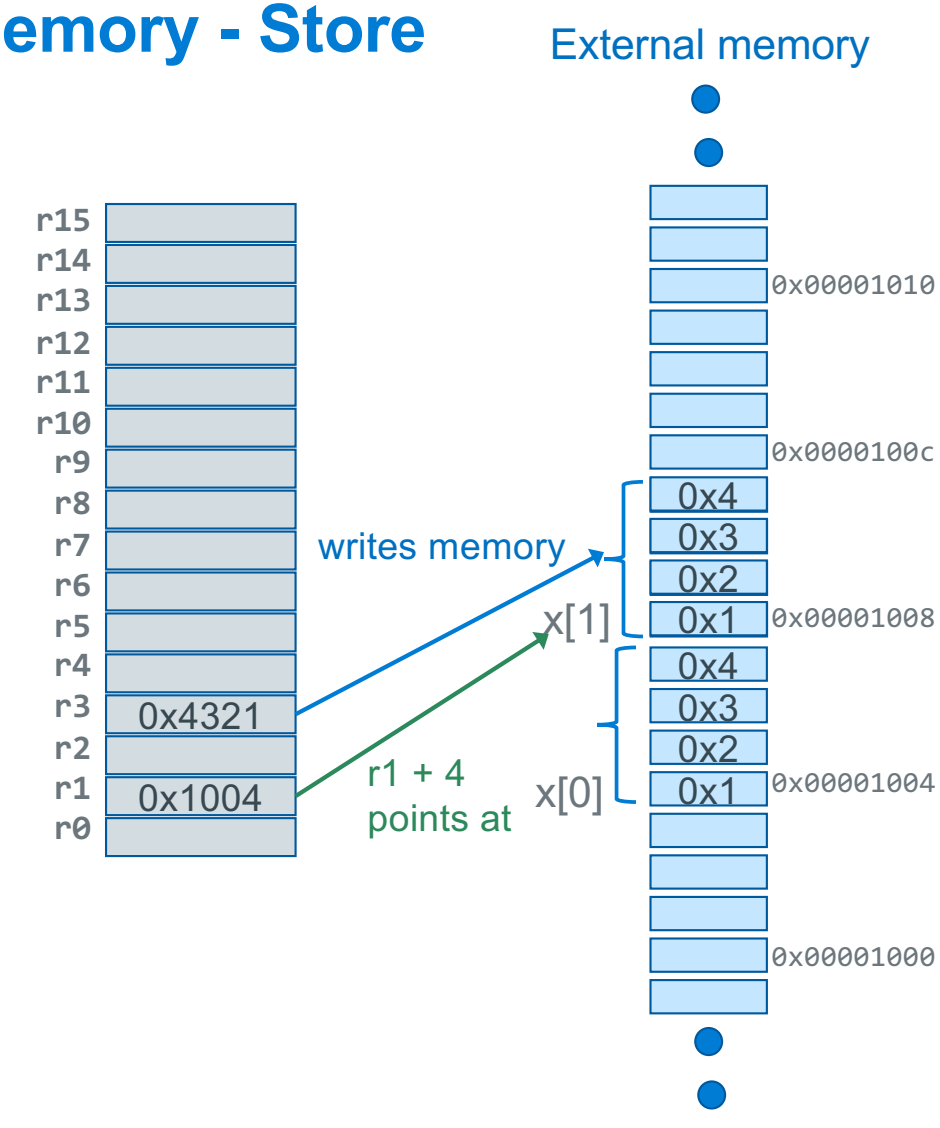
Using Registers as Pointers to Memory - Store

```
We want to do a x[1] = x[0]
We have to do this in two steps

int r3;
int x[2];
int *r1 = &x;     // r1 contains address
…
r3 = *(r1);       // memory to register
*(r1 + 1) = r3;   // register to memory
```

```
Store register to memory
str    r3, [r1, 4]
                ↑
address = r1 + 4 = 0x1008
```

The [ ] around the operands is like the * dereference op

We will cover this instruction in more detail later



x

Arm Register Summary

- 16 Named registers r0 – r15
- The operands of almost all instructions are registers
- To operate on a variable in memory do the following:
  1. Load the value(s) from memory into a register
  2. Execute the instruction
  3. Store the result back into memory (only if needed!)
- Going to/from memory is expensive
  - 4X to 20X+ **slower** than accessing a register
- Strategy: Keep variables in registers as much as possible

x

Using Aarch32 Registers

- There are two basic groups of registers, general purpose and special use
- General purpose registers can be used to contain up to 32-bits of data, but you must follow the **rules** for their use
  - Rules specify how registers are to be used so software can communicate and share the use of registers (later slides)
- Special purpose registers: dedicated hardware use (like r15 the pc) or special use when used with certain instructions (like r13 & r14)
- r15/pc is the program counter that contains the address of an instruction being executed (not exactly … later)

Special Use Registers
program counter

r15/pc

Special Use Registers
function call implementation
& long branching

r14/lr
r13/sp
r12/ip
r11/fp

Preserved registers
Called functions can't change

r10
r9
r8
r7
r6
r5
r4

Scratch Registers
First 4 Function Parameters
Function return value
Called functions can change

r3
r2
r1
r0

x

CPU Operational Overview: Executing Machine Code

Everything has a **memory address**: **instructions** & **data**

- Machine code uses addresses for loops, branches, function calls, variables, etc.

1 Fetch

- read the instruction into memory ***(fetch)***
- program counter is automatically incremented (+4) to contain the address of the next instruction in memory
  Instructions are 32 bits

2 Decode

- ***Decodes the instruction*** and sets up execution

3 Execute

- CPU completes the ***execution*** of the instruction
- Execution may alter the pc to take branches, etc.
- Go to **fetch**

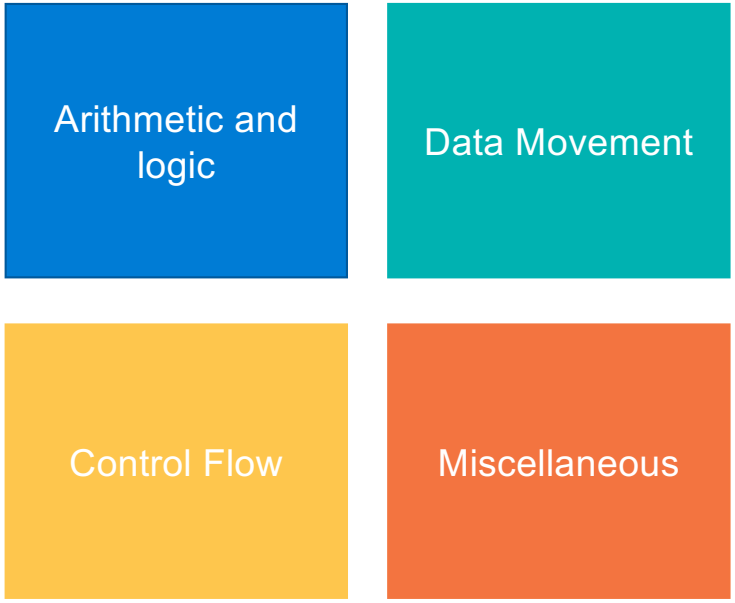
text segment in memory

	address	contents	assembly version
r15/pc	0001042c	<inloop>:	
r15/pc	1042c:	e3530061	cmp r3, 0x61
r15/pc	10430:	ba000002	blt 10440 <store>
r15/pc	10434:	e353007a	cmp r3, 0x7a
r15/pc	10438:	ca000000	bgt 10440 <store>
	1043c:	e2433020	sub r3, r3, #32
	00010440	<store>:	
r15/pc	10440:	e7c13002	strb r3, [r1, r2]
r15/pc	10444:	e2822001	add r2, r2, 0x1
	10448:	e7d03002	ldrb r3, [r0, r2]
	1044c:	e3530000	cmp r3, 0x0
	10450:	1afffff5	bne 1042c <inloop>

x

AArch32 Instruction Categories

- **Data movement to/from memory**
  - Data Transfer Instructions between memory and registers
    - Load, Store
- **Arithmetic and logic**
  - Data processing Instructions (registers only)
    - Add, Subtract, Multiply, Shift, Rotate, …
- **Control Flow**
  - Compare, Test, If-then, Branch, function calls
- **Miscellaneous**
  - Traps (OS system calls), Breakpoints, wait for events, interrupt enable/disable, data memory barrier, data synchronization barrier
  - Many others that we will not cover in the class



x

Basic Arm Machine Code Instructions

- Instructions consist of several fields that encode the opcode and arguments to the opcode
- Special fields enable extended functionality - later
- Several 4-bit operand fields for specifying the source and destination of the operation, usually one of the 16 registers
- Embedded constants *("immediate values")* of various size and "configuration"
- Basic Data processing instruction formats (below)
- R type instruction: `add r0, r1, r2` `// third operand is a register`
- I type instruction: `add r0, r0, 1` `// third operand is an immediate value`

	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	0	Rn	Rd	00000000	Rm

	4	3	4	1	4	4	4	8
I type:	1110	001	Opcode	0	Rn	Rd	Rotate	Imm

x

R (register) Type Data Processing: Machine Code

- Instructions that process data using three-register arguments
- The general instruction format is (not all fields will be in every instruction)

```
opcode   Rd (destination), Rn (operand 1), Rm (operand 2)
```

	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	0	Rn	Rd	00000000	Rm

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

```
add r0, r1, r3
```

is encoded as

1110 0000 1000 0001 0000 0000 0000 0011

in hex is

0xe0810003

x

I (immediate) Type Data Processing: Machine Code

- Instructions that process data using two registers and a constant (in the instruction)
- The general instruction format is (not all fields will be in every instruction)

```
opcode   Rd (destination), Rn (operand 1), constant
```

	4	3	4	1	4	4	4	8
type:	1110	001	Opcode	0	Rn	Rd	Rotate	Imm8

immediate value
12 bits only!!!!

0000 = AND - Rd:= Op1 AND Op2
0001 = EOR - Rd:= Op1 EOR Op2
0010 = SUB - Rd:= Op1 - Op2
0011 = RSB - Rd:= Op2 - Op1
0100 = ADD - Rd:= Op1 + Op2
0101 = ADC - Rd:= Op1 + Op2 + C
0110 = SBC - Rd:= Op1 - Op2 + C - 1
0111 = RSC - Rd:= Op2 - Op1 + C - 1
1000 = TST - set condition codes on Op1 AND Op2
1001 = TEQ - set condition codes on Op1 EOR Op2
1010 = CMP - set condition codes on Op1 - Op2
1011 = CMN - set condition codes on Op1 + Op2
1100 = ORR - Rd:= Op1 OR Op2
1101 = MOV - Rd:= Op2
1110 = BIC - Rd:= Op1 AND NOT Op2
1111 = MVN - Rd:= NOT Op2

```
add r0, r1, 49
```

is encoded as

1110 0000 1000 0001 0000 0000 0011 0001

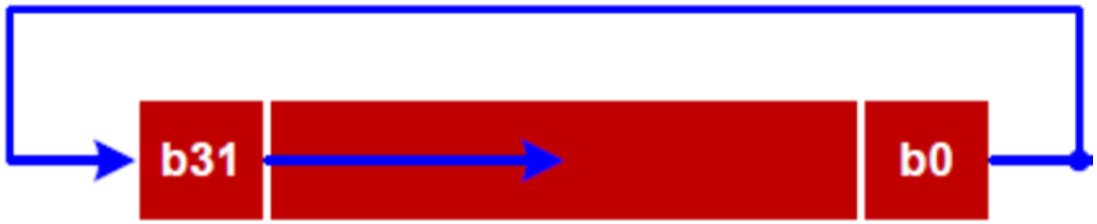
in hex is

0xe0810031

x

How are I – Type Constants Encoded in the instruction?

- Aarch32 provides only 8-bits for specifying an immediate constant value
- Without "rotation" immediate values are limited to the range of positive 0-255
- Imm8 expands to 32 bits and does a rotate right to achieve additional constant values (YUCK)



4	8
Rotate	Imm8
positions to **rotate X 2**	unsigned value to rotate

rot4 value	32-bit constant result
0000	0000 0000 0000 0000 0000 0000 1111 1111
0001 (2 bits)	1100 0000 0000 0000 0000 0000 0011 1111

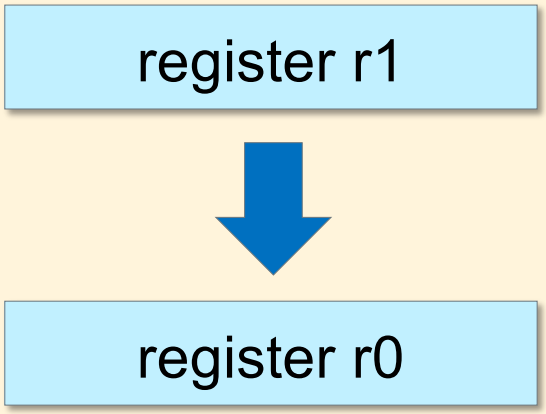
results are interpreted as a 2's complement number

two bits rotated right

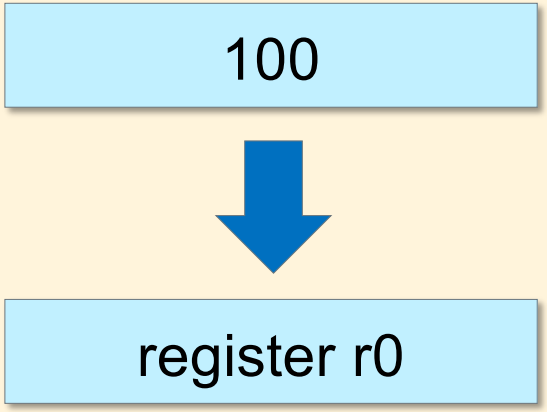
x

First Look: Copying Values To Registers - MOV

```
mov   r0, r1

// Copies all 32 bits
// of the value held
// in register r1 into
// the register r0
```

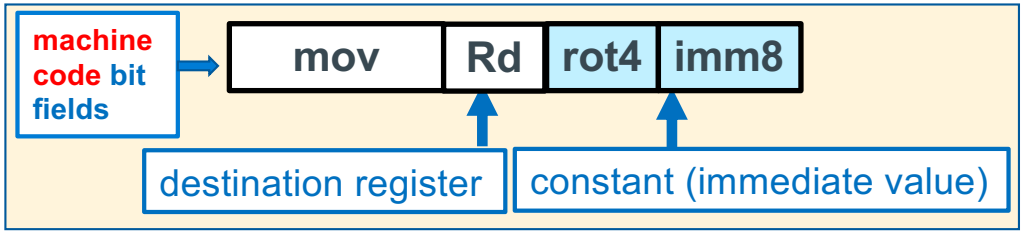


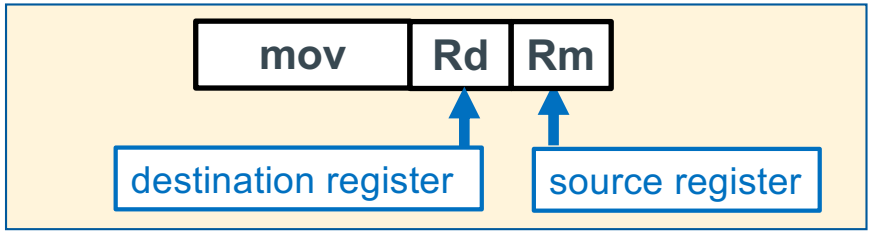

```
mov   r0, 100

// Expands an imm8 value 100
// stored in the instruction
// into the register r0
```



x

mov – Copies Register Content between registers





	4	3	4	1	4	4	8	4
R type:	1110	000	Opcode	S	0000	Rd	00000000	Rm
I type:	1110	001	Opcode	S	0000	Rd	Rotate	Imm8

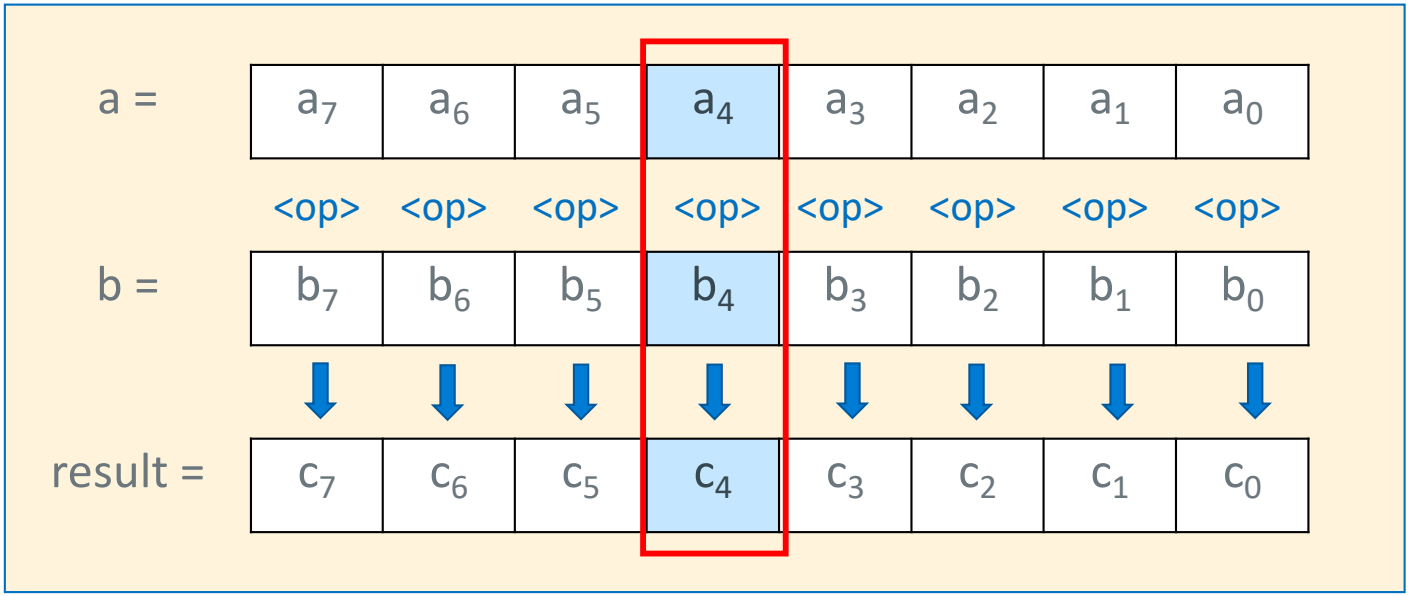
1101 - MOV
1111 - MVN

```
mov   Rd, constant    // Rd = constant
mov   Rd,  Rm         // Rd = Rm
```

```
mov   r1, r5          // r1 = r5
mov   r1, 1           // r1 = 1
mov   r1, -4          // r1 = -4
```


x

What is a Bitwise Operation?



- Bitwise operators are applied to each of the <u>corresponding</u> bit positions in each variable
- Each bit position of the result depends <u>only</u> on bits in the same bit position within the operands

x

Bitwise Not (vs Boolean Not)

```
in C
int output = ~a;
```

a	~a
0	1
1	0

Bitwise NOT

```
~ 1100
  ----
  0011
```

	Bitwise Not
number	0101 1010 0101 1010 1111 0000 1001 0110
~number	1010 0101 1010 0101 0000 1111 0110 1001

Meaning	*Operator*	*Operator*	*Meaning*
Boolean NOT	!b	~b	Bitwise NOT

Boolean operators act on the entire value not the individual bits

Type	*Operation*	result
bitwise	~0x01	1111 1111 1111 1111 1111 1111 1111 1110
Boolean	!0x01	0000 0000 0000 0000 0000 0000 0000 0000

x

First Look: Copying Values To Registers – MVN (negate)

```
mvn   r0, r1

// Copies all 32 bits
// of the value held
// in register r1 into
// the register r0
// then does a bitwise NOT
```

register r1

register r0

Bitwise NOT

```
~ 1100
  ----
  0011
```

- A **bitwise NOT** operation

```
mvn   r0, 12

// Expands an imm8 value 0x0c
// stored in the instruction
// into a register then does
// a bitwise NOT
```

register r0

0x0c

0xffff fff3

0x 0c

imm8 expansion

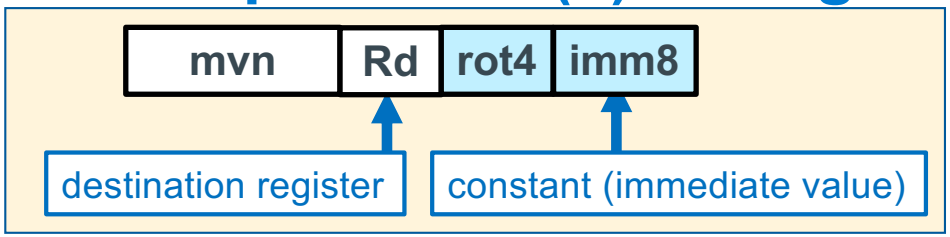
0x0000000c

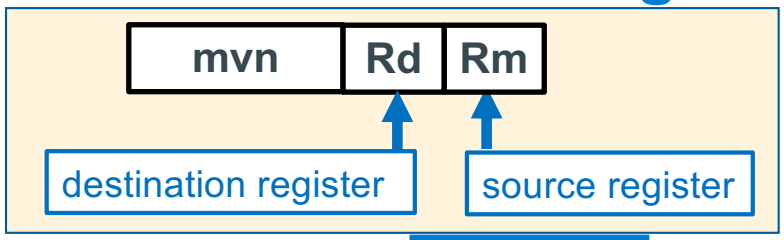
bitwise not

0xfffffff3

x

mvn – Copies NOT (~) of Register content between registers






```
mvn  Rd, constant    // Rd = constant
mvn  Rd,  Rm         // Rd = Rm
```

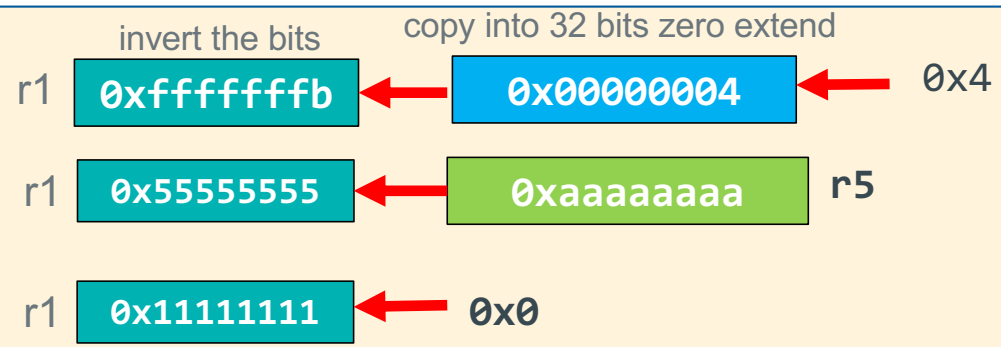
```
~ 1100
  ----
  0011
```

Bitwise NOT

bitwise NOT operation. Immediate (constant) version copies to 32-bit register, then does a bitwise NOT

imm8	extended imm8	inverted imm8	signed base 10
0x00	0x00 00 00 00	0xff ff ff ff	-1
0xff	0x00 00 00 ff	0xff ff ff 00	-256

```
mvn    r1, 4        // x = ~4
mvn    r1, r5       // x = ~y in C

mvn    r1, 0        // x = -1
```



x

First Look: Add/Sub Registers

```
add   r0, r1, r2
```

register r1 + register r2

```
// Adds r1 to r2 and
// stores the result
// in r0
```

register r0

```
sub   r0, r1, 100
```

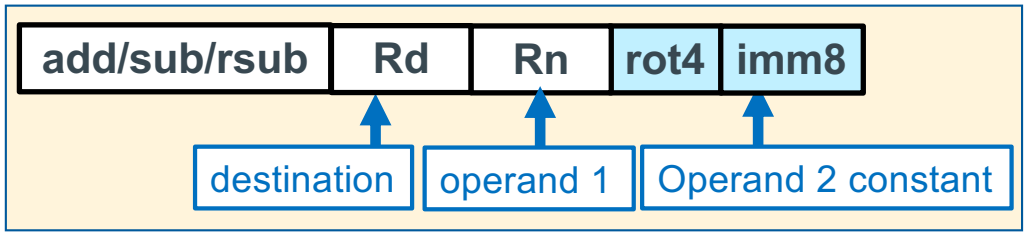
register r1 - 100

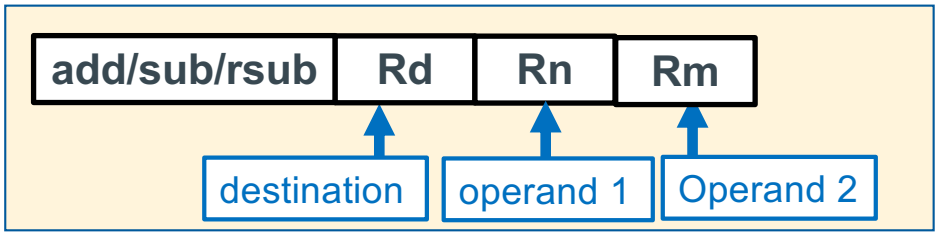
```
// Perform r1 – 100 and
// stores the result in
// r0
```

register r0

x

add/sub/rsub – Add or Subtract two integers






```
add  Rd, Rn, constant    // Rd = Rn + constant
sub  Rd, Rn, constant    // Rd = Rn - constant
rsub Rd, Rn, constant    // Rd = constant - Rn
add  Rd,  Rn,  Rm        // Rd = Rn + Rm
sub  Rd,  Rn,  Rm        // Rd = Rn - Rm
rsub Rd,  Rn,  Rm        // Rd = Rm - Rn
```

```
mov   r5, 5        // r5 = 5
mov   r7, 7        // r7 = 7
add   r7, r7, r5   // r7 = 12 r5 = 5
```

```
add   r1, r2, r3   // r1 = r2 + r3
sub   r1, r1, 1    // r1 = r1 – 1; or r1--
add   r1, r2, 234  // r1 = r2 + 234
```


x

Writing a Sequence of Add & Subtract Instructions

- You need to perform the following sequence of integer adds/subtracts

```
a = b + c + d - e;
```

- Since ARM uses a three-operand instruction set, you can only operate on two operands at a time
- So, you need to use one register as an **accumulator** and create **a sequence of add instructions** to build up the solution

```
r0 ← a
r1 ← b
r2 ← c
r3 ← d
r4 ← e
```

```
a  = b  + c  + d  - e;
r0 = r1 + r2 + r3 – r4;
r0 = ((r1 + r2) + r3) – r4;
r0 = r1 + r2;
r0 = r0 + r3
r0 = r0 – r4
```

```
add   r0, r1, r2
add   r0, r0, r3
sub   r0, r0, r4
```

```
a = (b + c) – 5;
r0 = (r1 + r2) – 5;
```

```
add   r0, r1, r2
sub   r0, r0, 5
```


x

Overview: Line Layout in an Arm Assembly Source File - 1

column 1 column 2 column 3 column 4

```
label:   operation operand(s)  // comments
```

tab white space white space

- Assembly language source text files are **line oriented** (each ending in a '\n')
- **Each line represents** a **starting address in memory** and does **one** of:
  1. Specifies the contents of memory for a variable (segments containing data)
  2. Specifies the contents of memory for an instruction (text segment)
  3. **Assembler directives** tell the assembler to do something (for example, change label scope, define a macro, etc.) that does not allocate memory
- **Each line** is **organized into up to four columns**
  - Not every column is used on each line
  - Not every line will result in memory being allocated

x

Overview: Line Layout in an Arm Assembly Source File - 2

```
label:  operation operand(s)  // comment

       // assembler directive below
cnt:    .word 5                /* define a global int cnt = 5;

        /* instruction below */
       add    r1   r2, r3       // add the values
```

1. **Labels** (optional); starts in column 1
   - **Only used** when you need to associate a name to a starting location in memory; You can then refer to the address **by name** in an instruction
2. **Operation type 1**: **assembler directives** (all start with a period e.g. `.word`)
3. **Operation Type 2**: assembly language **instructions**
4. **Zero or more operands** as required by the instruction or assembler directive
5. Comments C and C++ style; also @ in the place of a C++ comment //

x

Labels in Arm Assembly

local label

Regular label

local label

```
.Lmesg: .string  "Hello CSE30! We Are CountinG UpPpER cASe letters!"
...
main:
        push    {fp, lr}

...
.Lwhile:
        add     r2, r2, 1       // increment char pointer
```

label .Lmesg is the starting address for the ascii string

label main is the starting address of the push instruction

label .Lwhile is the starting address of the add instruction

- Remember, a Label associates a name with memory location
- **Regular Label:**
  - Used with a Function name (label) or for static variables in any of the data segments
- **Local Label:** Name starts with `.L` (local label prefix) only usable in the same file
  1. Targets for branches (if), switch, goto, break, continue, loops (for, while, do-while)
  2. Anonymous variables (string not foo in `char *foo = "anonymous variable"`)
  3. Read only literals when allocated in the text segment – special case)

x

Assembler Directives: Label Scope Control (Normal Labels only)

```
.extern printf
.extern fgets
.extern strcpy
.global fbuf
```

`.extern <label>`

- **Imports** label (function name, symbol or a static variable name);
- An address associated with the label from another file can be used by code in this file

`.global <label>`

- **Exports** label (or symbol) to be visible outside the source file boundary (other assembly or c source)
- label is either a function name or a global variable name
- **Without** .global, **by default** labels are local to the file from the point where they are defined

x

Assembler Directives: .equ and .equiv

```
.equ    BLKSZ,  10240      // buffer size in bytes
.equ    BUFCNT, 100*4      // buffer for 100 ints
.equiv  STRSZ,  128        // buffer for 128 bytes
.equiv  STRSZ,  1280       // ERROR! already defined!
.equ    BLKSZ,  STRSZ * 4  // redefine BLKSZ from here
```

`.equ <symbol>, <expression>`

- Defines and sets the value of a symbol to the evaluation of the expression
- Used for specifying constants, like a `#define` in C
- You can (re)set a symbol many times in the file, last one seen applies

```
.equ    BLKSZ,  10240        // buffer size in bytes
// other lines
.equ    BLKSZ,  1024         // buffer size in bytes
```

`.equiv <symbol>, <expression>`

- **`.equiv`** directive is like **`.equ`** except that the assembler will signal an error if symbol is already defined

x

Example: Assembler Directive and Instructions

assembler directive .equ does not allocate any memory (NULL = 0)

Regular label main is associated with memory location 0x3000

Local label .Lloop is associated with memory location 0x3004

space.S

```
10                          .equ    NULL, 0
11                  main:
12 3000 0310A0E1            mov     r1, r3
13                  .Lloop:
14 3004 043083E2            add     r3, r3, 4
15 3008 001093E5            ldr     r1, [r3]
16 300c 000051E3            cmp     r1, NULL
17 3010 FBFFFF1A            bne     .Lloop
```

Memory Contents
Warning contents shown in *"reverse"* byte order: Lsb – Msb

```
output generated with
gcc -c -Wa,-ahlns space.S
partial output is shown
```

Instruction Memory Addresses (lowest 2-bits are always are 00)
Notice alignment and how addresses increase by 4 (32-bit instructions)

x

Unconditional Branching – Forces Execution to Continue at a Specified Label (goto)

b	imm24

Unconditional Branch instruction *(branch to only local labels in CSE30)*

```
b .Llabel
```

- Causes an unconditional branch (aka goto) to the instruction with the address .`Llabel`
- `.Llabel` is called a **branch target label** (the *"target"* of a branch instruction)
- **Be careful! <u>do not to branch to a function label</u>!**
- `.Llabel`: pc is the base register with the offset being imm24 shifted left two bits (+/- 32 MB)
  - **imm24** is the **number of instructions** from **pc+8**

```
            b     .Lddone
                    :
.Ldone:     add    r0, EXIT_SUCCESS        // set return value
```


x

Examples of of Unconditional Branching

Unconditional Branch Forward

```
        b .Lforward
        add r1, r2, 4
        add r0, r6, 2
        add r3, r7, 4
.Lforward:
        sub r1, r2, 4
```

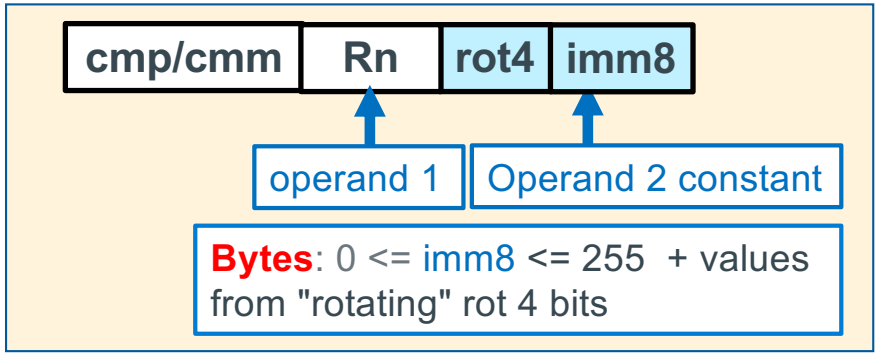
Infinite loop

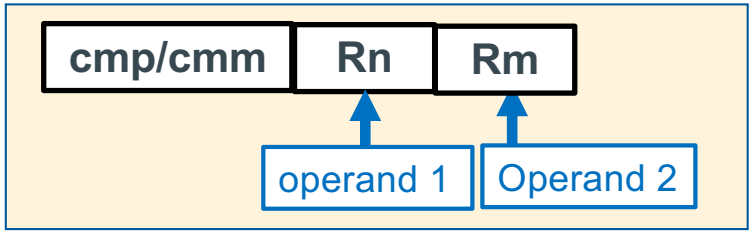
```
.Lbackward:
        add r1, r2, 4
        sub r1, r2, 4
        add r4, r6, r7
        b .Lbackward
       // not reachable
```

- Branches are used to change execution flow using labels as the branch target
- In these example, ***.Lforward*** and ***.Lbackward*** are the **branch target labels**
- Branch target labels are placed at the beginning of the line (or above it)

x

cmp/cmm – Making Conditional Tests






```
cmp   Rn, constant      // Rn – constant; then sets condition flags
cmm   Rn, constant      // Rn + constant; then sets condition flags
cmp   Rn,  Rm           // Rn – Rm; then sets condition flags
cmm   Rn,  Rm           // Rn + Rm; then sets condition flags
```

The values stored in the registers Rn and Rm are not changed
The assembler will automatically substitute `cmn` for negative immediate values

```
cmp    r1, 0        // r1 – 0 and sets flags on the result
cmp    r1, r2       // r1 – r2 and sets flags on the result
```


x

Quick Overview of the Condition Bits/Flags

- The CSPR is a special register (like the other registers) in the CPU
- The four bits at the left are called the Condition Code flags
  - **Summarize** the **result** of a previous instruction
  - Not all instruction will change the CC bits
- **Specifically, Condition Code flags are set** by cmm/cmp (and others)

```
Example:            cmp    r4, r3
```

- **N** (Negative) **flag**: Set to 1 when the result of r4 – r3 is negative, set to 0 otherwise
- **Z (Zero) flag:** Set to 1 when the results of r4 – r3 is 0, set to 0 otherwise
- **C (Carry bit) flag:** Set to 1 when r4 – r3 does not have a borrow, set to 0 otherwise
- **V flag** (oVerflow): Set to 1 when r4 – r3 causes an overflow, set to 0 otherwise

x

Conditional Branch: Changing the Next Instruction to Execute

cond	b	imm24

Branch instruction

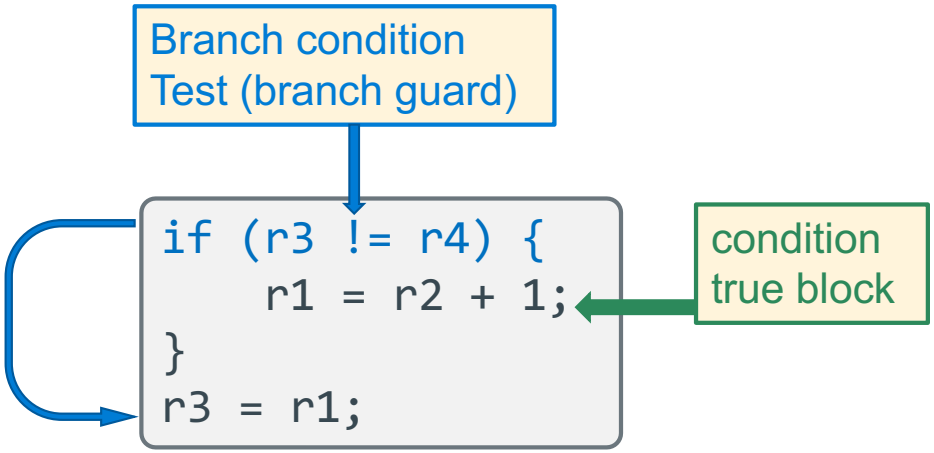
```
bsuffix .Llabel
```

- Bits in the condition field specify the **conditions** when the branch happens
- If the condition evaluates to be true, the next instruction executed is located at `.Llabel:`
- If the condition evaluates to be false, the next instruction executed is located immediately after the branch
- Unconditional branch is when the condition is *"always"*

Condition	*Meaning*	*Flag Checked*
BEQ	Equal	Z = 1
BNE	Not equal	Z = 0
BGE	Signed ≥ ("Greater than or Equal")	N = V
BLT	Signed < ("Less Than")	N ≠ V
BGT	Signed > ("Greater Than")	Z = 0 && N = V
BLE	Signed ≤ ("Less than or Equal")	Z = 1 || N ≠ V
BHS	Unsigned ≥ ("Higher or Same") or Carry Set	C = 1
BLO	Unsigned < ("Lower") or Carry Clear	C = 0
BHI	Unsigned > ("Higher")	C = 1 && Z = 0
BLS	Unsigned ≤ ("Lower or Same")	C = 0 || Z = 1
BMI	Minus/negative	N = 1
BPL	Plus - positive or zero (non-negative)	N = 0
BVS	Overflow	V = 1
BVC	No overflow	V = 0
B (BAL)	Always (unconditional)	

x

Anatomy of a Sample Branch: Changing the Next Instruction to Execute



- **Branch guard**: determines whether to execute the "true" block
- When the branch guard evaluates to false, branch around the true block

x

Conditional Branch: Changing the Next Instruction to Execute

```
        cmp     r3, r4
        beq     .Ldone
        // otherwise do the add
        add     r1, r2, 1
        b       .Ldone
.Ldone:
        mov     r3, r1
```

```
if (r3 != r4)
    r1 = r2 + 1;
r3 = r1;
```

Condition	Meaning	Flag Checked
BEQ	Equal	Z = 1
B	Always (unconditional)	

```
cmp     r3, r4 // r3 – r4
// if r3 != r4 sets Z = 0
```

1. Use a **cmp/cmm** instruction to set the condition bits
2. Follow the **cmp/cmm** **with one or more variants of the conditional branch instruction**
   Conditional branch instructions if evaluate to true (bases on the CC bits set) will go to the instruction with the branch label. Otherwise, it executes the instruction that follows

- You can have one or more conditional branches after a single cmp/cmm

x

When do you use a Signed or Unsigned Conditional Branch?

example with 4 bits
(arm is 32)

$0111_2 > 1100_2$?

unsigned → Compare 7 > 12 ? **(FALSE)**

signed 2's complement → Compare +7 > -4 ? **(TRUE)**

Condition	*Suffix For Unsigned Operands:*	*Suffix For Signed Operands:*
>	**BHI** *(Higher Than)*	**BGT** *(Greater Than)*
>=	**BHS** *(Higher Than or Same) (BCS)*	**BGE** *(Greater Than or Equal)*
<	**BLO** *(Lower Than) (BCC)*	**BLT** *(Less Than)*
<=	**BLS** *(Lower Than or Same)*	**BLE** *(Less Than or Equal)*
==	**BEQ** *(Equal)*	
!=	**BNE** *(Not Equal)*	

x

Review Anatomy of a Conditional Branch: If statement

Branch condition
Test (branch guard)

```
if (r0 == 5) {
      /* condition block #1 */
} else {
      /* condition block #2 */
}
```

condition
true block

condition
false block

- **In C**, when the branch guard (condition test) evaluates **non-zero** you ***fall through*** to the ***condition true*** block, otherwise you branch to the *condition false* block
- Block order: (the order the **blocks appear** in C code) can be changed by **inverting** the conditional test, **swapping** the order of the true and false blocks

Branch condition
Test (branch guard)

```
if (r0 != 5) {
      /* condition block #2 */
} else {
      /* condition block #1 */
}
```

condition
true block

condition
false block

x

Examples: Guards (Conditional Tests) and their Inverse

Compare in C	*"Inverse"* Compare in C
`==`	`!=`
`!=`	`==`
`>`	`<=`
`>=`	`<`
`<`	`>=`
`<=`	`>`

- Changing the conditional test (guard) to its inverse, allows you to swap the order of the blocks in an if else statement

x

Program Flow: Keeping the same *"Block Order"* as C

Branch guard

```
int r0;
if (r0 == 5) {
    /* when true "fall through" to here */
    /* condition true block */
}
/* in assembly branch to here if not true */
```

condition true block

- In ARM32, you either **fall through** (execute the *next instruction in sequence*) or **branch to a specific instruction** and then *resume* sequential instruction execution
- In order to keep the **same block order** as the **C version** that says: ***fall through*** to the condition **true** block when the branch guard evaluates to be **true**
  - Assembly: ***invert*** the **condition test** to **branch around** the ***condition true*** block
- Summary: In ARM32 use a **condition test** that **specifies the <u>opposite</u> of the condition used in C , then branch around** the **condition true** block

x

Branch Guard "*Adjustment*" Table Preserving Block Order In Code

Compare in C	"*Inverse*" Compare in C	"*Inverse*" Signed Assembly	"*Inverse*" Unsigned Assembly
==	!=	bne	bne
!=	==	beq	beq
>	<=	ble	bls
>=	<	blt	blo
<	>=	bge	bhs
<=	>	bgt	bhi

```
if (r0 compare 5)
    /* condition true block */
}
```

```
    cmp r0, 5
    inverse .Lelse
    // condition true block
.Lendif:
```


x

Program Flow: Simple If statement, No Else

Approach: adjust the conditional test then **branch around** the **true block**

Use a **conditional test** that **specifies the inverse of the condition used in C**

C source Code	*Incorrect Assembly*	*Correct Assembly*

`int r0;`
`if (r0 > 10)`
	`cmp r0, 10`
	`bgt .Lendif`
	
	`.Lendif:`
		`cmp r0, 10`
		`ble .Lendif`
		
		`.Lendif:`

```
int r0;
if (r0 == 5) {
    /* condition true block */
    /* fall through */
}
/* branch around to this code */
```

If r0 == 5 true then ***fall through*** to the true block

```
    cmp r0, 5
    bne .Lendif
    /* condition true block */
    /* fall through */
.Lendif:
/* branch around to this code */
```

If r0 == 5 false then branch ***around*** the true block

x

If statement examples – Branch Around the True block!

```
int r0;
if (r0 == 5) {
    r1 = r2++ + r3;
}
r2 = r3;
```

```
    cmp   r0, 5
    bne   .Lendif
    add   r1, r2, r3
    add   r2, r2, 1
.Lendif:
    mov   r3, r2
```

If r0 == 5 false then branch ***around*** the true block

```
int r0;
if (r0 <= 5) {
    r1 = r2++;
}
r2 = r3;
```

```
    cmp   r0, 5
    bgt   .Lendif
    mov   r1, r2
    add   r2, r2, 1
.Lendif:
    mov   r3, r2
```

```
unsigned int r0, r1;
if (r0 > r1) {
    r1 = r0;
}
r2 = r3;
```

```
    cmp   r0, r1
    bls   .Lendif
    mov   r1, r0
.Lendif:
    mov   r3, r2
```


x

Program Flow: If with an Else

```
if (r0 == 5) {
    /* cond. true block */
} else {
    /* condition false block */
    /* fall through */
}
r1 = 4;
```

If r0 == 5 false then branch ***to*** false block

```
    cmp r0, 5
    bne .Lelse
    /* cond. true block */

    /*
     * Now branch around the
     * condition false block
     */
    b .Lendif
.Lelse
    /* condition false block */
    /* fall through */
.Lendif:
    mov r1, 4
```

If r0 == 5 false then branch ***to*** false block

1. Make the adjustment to the conditional test to **branch to** the false block
2. When you finish the true block, you do an unconditional branch around the false block
3. The false block *falls through* to the following instructions

x

If with an Else Examples

Branch condition
Test (branch guard)

```
…
if (r0 < r1) {
    r2 = 1;
    // branch around else
} else {
    r2 = 0;
    /* fall through */
}
r4 = r2 + r4;
```

```
        cmp r0, r1
        bge .Lelse

        mov r2, 1
        b .Lendif
.Lelse:
        mov   r2, 0
        /* fall through */

.Lendif:
        add   r4, r2, r4
```

If r0 < r1 false
then branch
to false block

condition
true block

condition
false block

x

If with an Else Block order: All These Are Equivalent

```
if (r0 < r1) {
   r2 = 1;
   // branch around else
} else {
   r2 = 0;
   /* fall through */
}
r4 = r2 + r4;
```

```
      cmp r0, r1
      bge .Lelse
      mov r2, 1
      b .Lendif
.Lelse:
      mov   r2, 0
     /* fall through */
.Lendif:
      add   r4, r2, r4
```

```
if (r0 >= r1) {
    r2 = 0;
    // branch around else
} else {
   r2 = 1;
   /* fall through */
}
r4 = r2 + r4;
```

```
      cmp r0, r1
      blt .Lelse
      mov   r2, 0
      b .Lendif
.Lelse:
      mov r2, 1
     /* fall through */
.Lendif:
      add   r4, r2, r4
```


x

Branching What not to do: Spaghetti Code

```
.Lelse:
        mov    r2, 0
        b      .Lendif // not needed, slows code
.Lendif:
        add    r1, r2, r3
```

Use fall-through!
do not branch to the next statement!

```
.Lelse:
        mov    r2, 0
        add    r1, r2, r3
```

```
        mov    r1, 1
        mov    r2, 2
        b      .Lthree
        mov    r5, 5
        b      .Lsix
.Lthree:
        mov    r3, 3
        mov    r4, 4
        b      .Lseven
.Lsix:
        mov    r6, 6
.Lseven:
        mov    r7, 7
```

Observation
Using **many br** commands is a sign you should look to reorganize your code

Much faster and easier to read!

```
        mov    r1, 1
        mov    r2, 2
        mov    r3, 3
        mov    r4, 4
        b      .Lseven
        mov    r5, 5
        mov    r6, 6
.Lseven:
        mov    r7, 7
```

Notice after *"unwinding"* this *unreachable code* is easier to detect

x

Bad Style: Branching Upwards (Not a loop)

Do not Branch "Upwards" unless it is part of a loop (later slides)

- If you cannot easily write the equivalent C code for your assembly code, you may have code that is harder to read than it should be
- Action: adjust your assembly code to have a similar structure as an equivalent version written in C

```
        cmp r3, r4
        beq .Lelse1
        bgt .Lelse2
        cmp r0, r1
        bge .Lelse2
        mov r2, 1
        b .Lendif
.Lelse1:
        mov   r2, 0
         b .Lendif
.Lelse2:
         add r4, r3, r2
         b .Lesle1
.Lendif:
         add   r4, r2, r4
```


x

Branching: Use Fall through!

Avoid goto like structure

- Do not use unnecessary branches (sometimes called a “goto” like structure) when a “fall through” works
- You can see this by structures that have a **conditional branch around an unconditional branch that immediately follows it**

Do not do the following:

```
    cmp r0, 0
    beq .Lthen
    b .Lendif
.Lthen:
    add r1, r1, 1
.Lendif:
    add r1, r1, 2
```

Not good!
Two adjacent branches

Do the following:

```
    cmp r0, 0
    bne .Lendif  // fall through
    add r1, r1, 1
.Lendif:
    add r1, r1, 2
```


x

Switch Statement

```
switch (r0) {
case 1:
    // block 1
    break;
case 2:
    // block 2
    break;
default:
    // default 3
    break;
}
```

Approach 1 – *Branch Block*

```
    cmp r0, 1
    beq .Lblk1
    cmp r0, 2
    beq .Lblk2

    // fall through
    // default 3
    b .Lendsw // break
.Lblk1

    // block 1
    b .Lendsw // break
.Lblk2:

    // block 2
    // fall through
    // NO b .Lendsw
.Lendsw:
```

Branch block

Approach 2 – if else equiv.

```
    cmp r0, 1
    bne .Lblk2

    // block 1
    b .Lendsw // break
.Lblk2:
    cmp r0, 2
    bne .Ldefault

    // block 2
    b .Lendsw // break
.Ldefault:

    // default 3
    // fall through
    // NO b .Lendsw
.Lendsw:
```


x

Program Flow – Short Circuit or Minimal Evaluation

- In evaluation of conditional guard expressions, C uses what is called **short circuit** or **minimal** evaluation

```
if ((x == 5) || (y > 3))  // if x == 5 then y > 3 is not evaluated
```

- **Each** expression argument is evaluated **in sequence** from left to right including any side effects (modified using parenthesis), **before** (optionally) evaluating the next expression argument

```
if (x || ++x) // true block always executed: ++x!
    printf("%d\n", x);
```

- If after evaluating an argument, the value of the entire expression can be determined, then the remaining arguments are NOT evaluated *(for performance)*

```
if ((a != 0) && func(b))     // if a is 0, func(b) is not called
    // do_something();
```


x

Program Flow – If statements && compound tests - 1

```
if ((r0 == 5) && (r1 > 3)) {
    r2 = r5;  // true block
    /* fall through */
}
r4 = r3;
```

```
    cmp r0, 5
    bne .Lendif

    cmp r1, 3
    ble .Lendif
    mov r2, r5. // true block
    // fall through
.Lendif:
    mov r4, r3
```

If r0 == 5 false then short circuit branch ***around*** the true block

x

Program Flow – If statements && compound tests - 2

```
if ((r0 == 5) && (r1 > 3))
{
    r2 = r5; // true block
    // branch around else
} else {
    r5 = r2; False block */
    /* fall through */
}
r4 = r3;
```

```
    cmp r0, 5  // test 1
    bne .Lelse

    cmp r1, 3  // test 2
    ble .Lelse

    mov r2, r5 // true block
    // branch around else
    b .Lendif
.Lelse:
    mov r5, r2 //false block
    // fall through
.Lendif:
   mov r4, r3
```

if r0 == 5 false then short circuit branch ***to*** the false block

if r1 > 3 false then branch ***to*** the false block

x

Program Flow – If statements || compound tests - 1

```
if ((r0 == 5) || (r1 > 3)) {
    r2 = r5; // true block
    /* fall through */
}
r4 = r3;
```

```
    cmp r0, 5
    beq .Lthen

    cmp r1, 3
    ble .Lendif
    // fall through
.Lthen:
    mov r2, r5 // true block
    /* fall through */
.Lendif:
    mov r4, r3
```

If r0 == 5 true, then branch ***to*** true block

if r1 > 3 false then branch ***around*** true block

x

Program Flow – If statements || compound tests - 2

```
if ((r0 == 5) || (r1 > 3)) {
    r2 = r5; // true block
    /* branch around else */
} else {
    r5 = r2; // false block
    /* fall through */
}
```

```
    cmp r0, 5
    beq .Lthen

    cmp r1, 3
    ble .Lelse
    // fall through
.Lthen:
    mov r2, r5 // true block
    // branch around else
    b .Lendif
.Lelse
    mov r5, r2 // false block
    // fall through
.Lendif:
```

If r0 == 5 true, then branch ***to*** the true block

if r1 > 3 false then branch ***to*** false block

x

Program Flow – multiple branches, one cmp

```
if ((r0 > 5) {
    /* condition block 1 */
    // branch to endif
} else if (r0 < 5){
    /* condition block 2 */
    // branch to endif
} else {
    /* condition block 3 */
    // fall through to endif
}
// endif
r1 = 11;
```

```
    cmp r0, 5
    bgt .Lblk1
    blt .Lblk2
    // fall through
    // condition block 3
    b .Lendif
.Lblk1:
    // condition block 1
    b .Lendif
.Lblk2:
    // condition block 2
    b .Lendif
.Lendif:
    mov r1, 5
```

special case: multiple branches from one cmp

- There are many other ways to do this

x

Program Flow – Pre-test and Post-test Loop Guards

- loop guard: code that must evaluate to true before the next iteration of the loop
- If the loop guard test(s) evaluate to true, the *body of the loop* is executed again
- pre-test loop guard is at top of the loop
  - If the test evaluates to true, execution falls through to the loop body
  - if the test evaluates to false, execution **branches** around the loop body
- post-test loop guard is at the bottom of the loop
  - If the test evaluates to true, execution **branches** to the top of the loop
  - If the test evaluates to false, execution falls through the instruction following the loop

zero or more iterations

```
while (i < 10) {
    /* block */
    i++;
}
```

pre-test loop guard

loop control variable

one or more iterations

```
do {
    /* block */
    i++;
} while (i < 10);
```

post-test loop guard

x

Pre-Test Guards - While Loop

```
while (r1 < 10) {
    /* block */
    r1++;
}
r2 = r1;
```

```
.Lwhile:
    cmp r1, 10
    bge .Lendw
    // block
    add r1, r1, 1
    b .Lwhile
.Lendw:
    mov r2, r1
```

pre-test loop guard

loop iteration

inverted test

```
while (r1 < 10) {
    if (r2 != r1) {
        /* block */
    }
    r1++;
}
r2 = r1;
```

```
.Lwhile:
    cmp r1, 10
    bge .Lendw
    cmp r2, r1
    beq .Lnext
    // block
.Lnext:
    add r1, r1, 1
    b .Lwhile
.Lendw:
    mov r2, r1
```

pre-test loop guard

loop iteration

x

Post-Test Guards – Do While Loop

```
do {
    /* block */
    r1++;
} while (r1 < 10);

r2 = r1;
```

loop iteration

post-test loop guard

```
.Ldo:
    // block
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo

    mov r2, r1
```

test is not inverted

```
do {
    if (r2 != r1) {
        /* block */
    }
    r1++;
} while (r1 < 10);

r2 = r1;
```

loop iteration

post-test loop guard

```
.Ldo:
    cmp r2, r1
    beq .Lnext
    // block
.Lnext
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo

    move r2, r1
```


x

Program Flow – Counting (For) Loop

Pre loop setup

Pre-test Loop guard

Post loop update

```
for (r1 = 0; r1 < 10; r1++) {
    /* block */
}
```

Pre loop setup

Pre-test loop guard

Post loop update

```
    mov r1, 0
.Lfor:
    cmp r1, 10
    bge .Lendfr
    // block code
    add r1, r1, 1
    b .Lfor
.Lendfr:
```

A **counting loop** has three parts:

1. Pre-loop setup
2. Pre-test loop guard conditions
3. Post-loop update

- Alternative:
- move Pre-test loop guard before the loop
- Add post-test loop guard
  - *converts* to *do while*
  - **removes** an **unconditional branch**

Alternative

Pre-loop setup

Pre-test loop guard

Post loop update

Post-test loop guard

```
    mov r1, 0
    cmp r1, 10
    bge .Lendfr
.Ldo:
    // block code
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo
.Lendfr:
```


x

Nested loops

```
for (r3 = 0; r3 < 10; r3++) {
    r0 = 0;

    do {
        r0 = r0 + r1++;
    } while (r1 < 10);

    // fall through
    r2 = r2 + r1;
}
r5 = r0;
```

```
    mov r3, 0
.Lfor:
    cmp r3, 10      // loop guard
    bge .Lendfor

    mov r0, 0

.Ldo:
    add r0, r0, r1
    add r1, r1, 1

    cmp r1, 10   // loop guard
    blt .Ldo

    // fall through
    add r2, r2, r1

    add r3, r3, 1 // loop iteration
    b .Lfor
.Lendfor:
    mov r5, r0
```

- Nest loop blocks as you would in C or Java
- Do not branch into the middle of a loop, this is hard to read and is prone to errors

x

Keeps loops Properly Nested

- It is hard to understand and debug loops when the "branch into each other"
- **Keep loops proper nested**

Do not do the following:

```
.Lloop1:
    add r1, r1, 1
.Lloop2:
    add r2, r2, 1
    add r2, r1, r3
    cmp r1, 10
    blt .Lloop1
    beq .Lend1
    add r3, r3, 1
    cmp r2, 20
    ble .Lloop2
.Lend1:
```


x

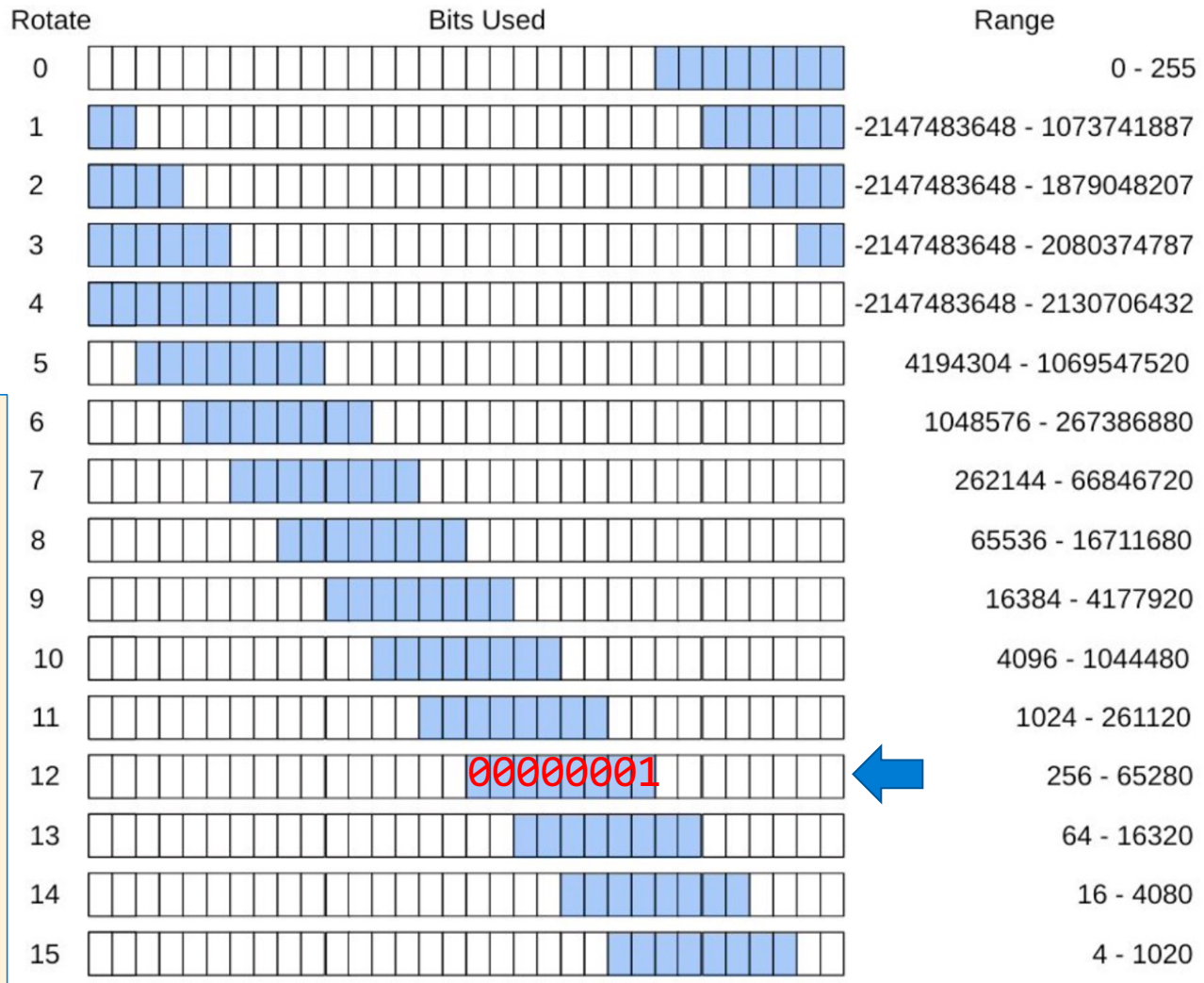
more to come….

Extra Slides Information only Not part of class

Rot4 - Imm8 Values

4	8
Rotate	Imm8
positions to rotate X 2	unsigned value to rotate

- How would 256 be encoded?
  - rotate = 12, imm8 = 1
- **Bottom line:** the assembler will do this for you
- If you try and use an immediate value that it cannot generate it will give an error
- There is a workaround - later



results are interpreted as a 2's complement number

x

Branch Target Address (BTA): What Is imm24?

executing instruction

decode instruction

fetch instruction

```
0001042c <inloop>:
   1042c: e3530061    cmp r3, 0x61
   10430: ba000002    blt 10440 <store>
   10434: e353007a    cmp r3, 0x7a
   10438: ca000000    bgt 10440 <store>
   1043c: e2433020    sub r3, r3, #32

00010440 <store>:
   10440: e7c13002    strb r3, [r1, r2]
   10444: e2822001    add r2, r2, 0x1
   10448: e7d03002    ldrb r3, [r0, r2]
   1044c: e3530000    cmp r3, 0x0
   10450: 1affffff5   bne 1042c <inloop>
```

BTA: + 2 instructions

- Previous slide: phases of execution: (1) fetch, (2) decode, (3) execute
- The pc (r15) contains the address of the instruction being fetched, which is two instructions ahead or executing instruction + 8 bytes
- **Branch target address** (or imm24) is the distance measured in the # of instructions (signed, 2's complement) from the fetch address contained in r15 when executing the branch

```
target address    = 0x10440
fetch address     = 0x10438
distance(bytes)   = 0x00008
distance(instructions)= 0x8/(4 bytes/instruction)= 0x2
```

imm24	0x 00 00 02

x