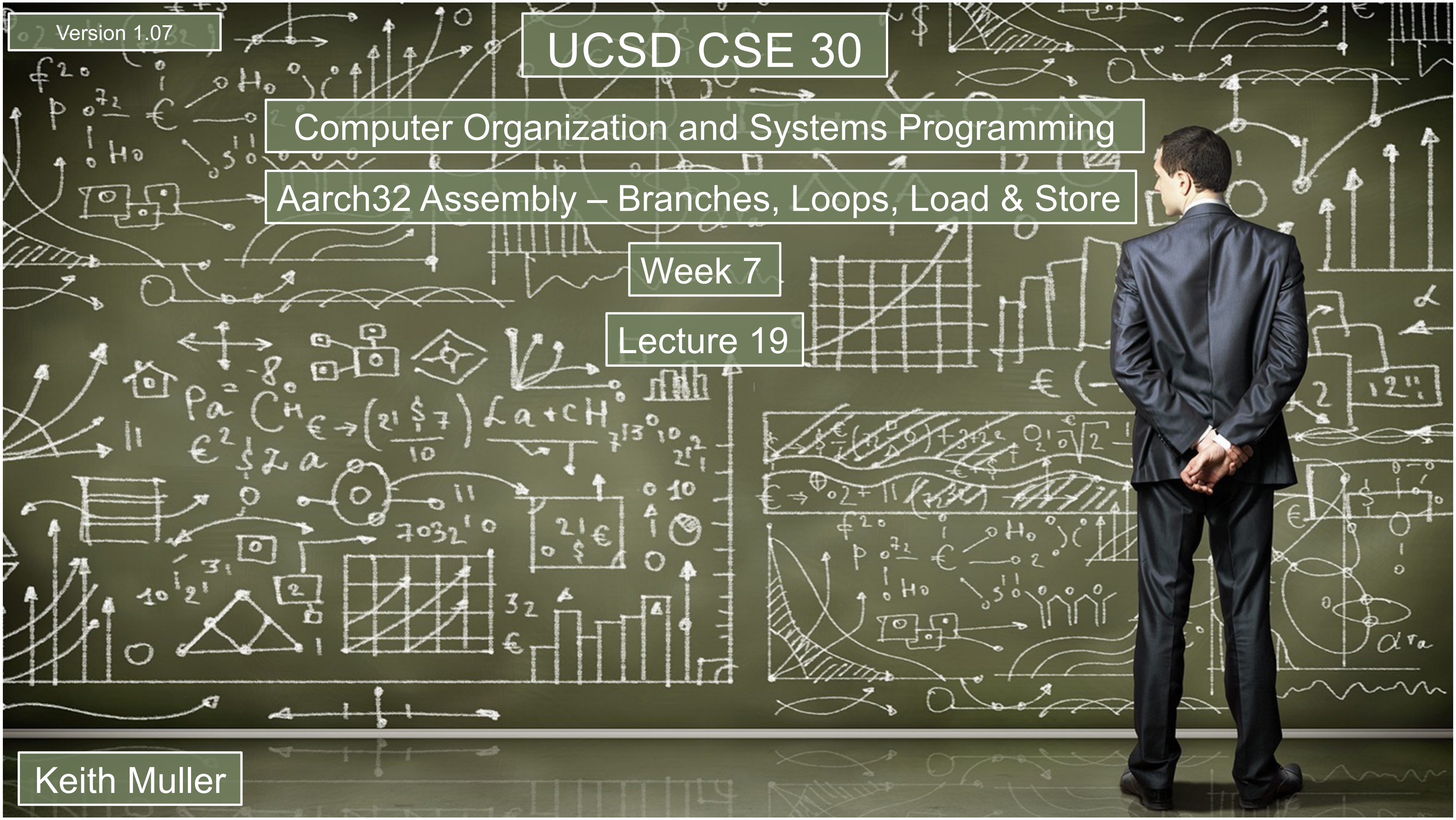


Version 1.07

UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Branches, Loops, Load & Store

Week 7

Lecture 19

Keith Muller

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: .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)

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
```

Regular label main is associated with memory location 0x3000

Local label .Lloop is associated with memory location 0x3004

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

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

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


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! do not to branch to a function label!**
- `.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     .Ldone
                    :
.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

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

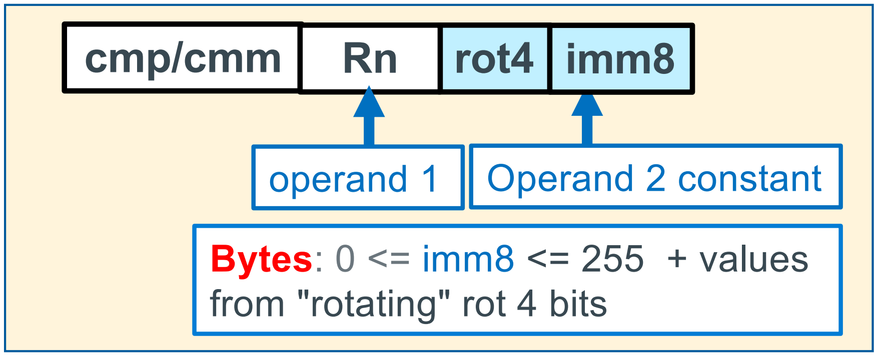
Examples: Guards (Conditional Tests) and their Inverse

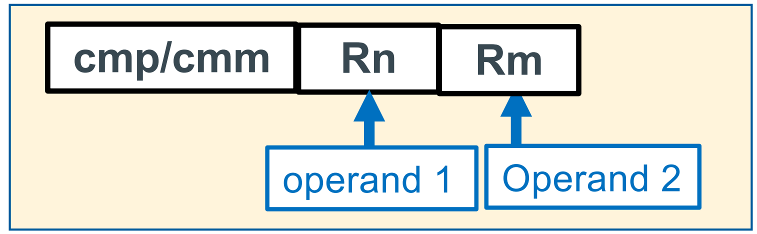
`Compare in C`	`"Inverse" Compare in C`
`==`	`!=`
`!=`	`==`
`>`	`<=`
`>=`	`<`
`<`	`>=`
`<=`	`>`

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

x

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
```

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

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

Conditional Branch: Changing the Next Instruction to Execute

```
      cmp    r3, r4
      beq   .Ldone
      blt   .Lelse
      // otherwise do the add
      add   r1, r2, 1
      b     .Ldone
.Lelse:
      sub   r0, r0, r1
.Ldone:
      mov   r3, r1
```

Condition	Meaning	Flag Checked
BEQ	Equal	Z = 1
BLT	Signed < ("Less Than")	N ≠ V
B	Always (unconditional)	

```
cmp    r3, r4 // r3 - r4
// if r3 == r4 sets Z = 1
// if r3 < r4 sets N and V; is N == V?
```

Two steps to do a Conditional Branch: Use a **cmp/cmm** instruction to set the condition bits

1. 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
2. You can have one or more conditional branches after a single cmp/cmm

x

When do you use a Signed or Unsigned Conditional Branch?

Compare
7 > 12 ?
(FALSE)

unsigned

4-bit compare
$0111_2 > 1100_2$?

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

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: 1afffff5    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

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

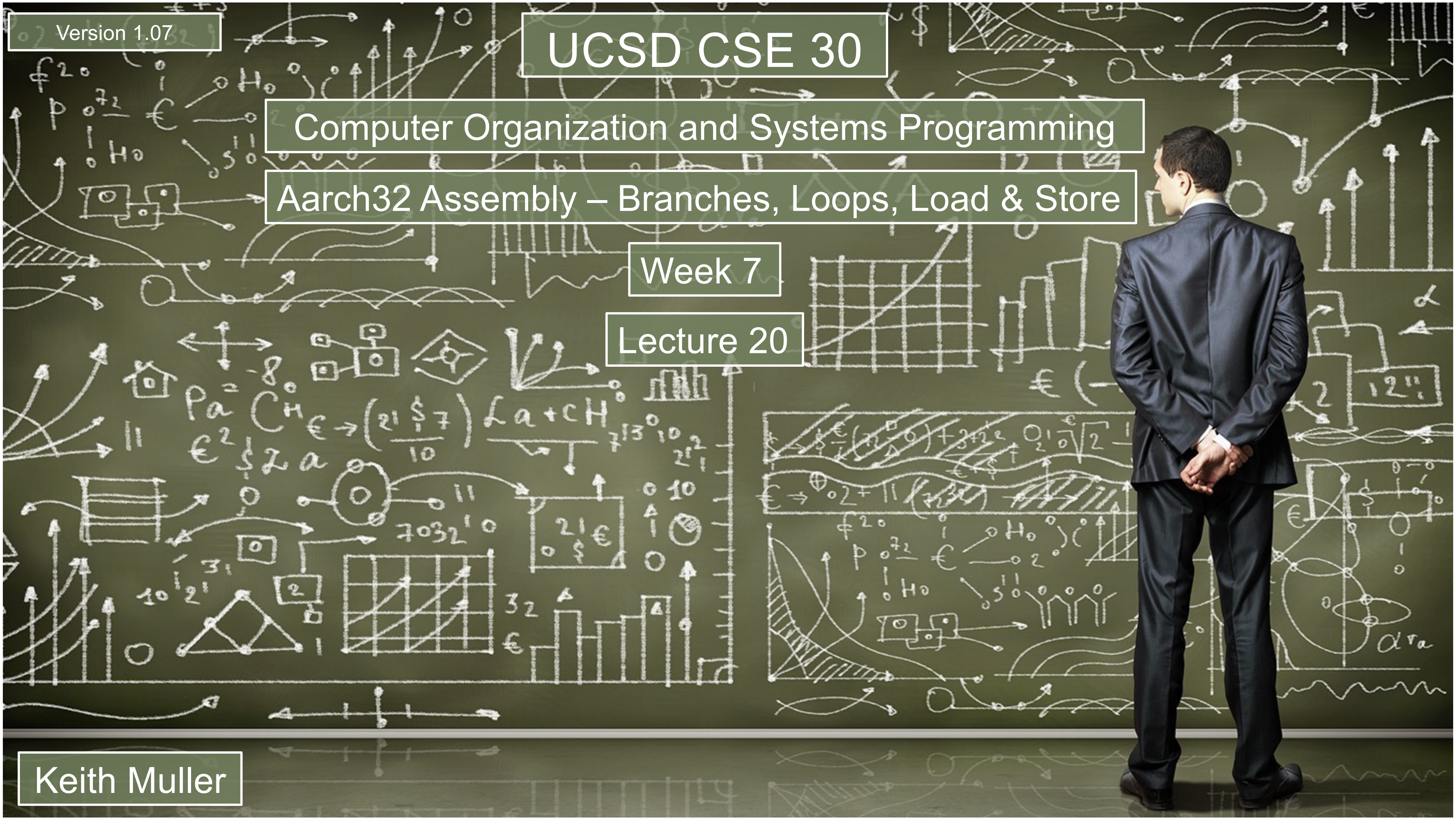
```
    cmp r0, 5
    bne .Lelse
    /* cond. true block */

    /*
     * 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



Version 1.07

UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Branches, Loops, Load & Store

Week 7

Lecture 20

Keith Muller

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 opposite of the condition used in C , then branch around** the **condition true** block

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

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

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

```
.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

Branching: What Not to Do

Guidelines

- 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

Switch Statement

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

Approach 1 – *Branch Block*

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:
```

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;
```

- There are many other ways to do this

```
    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

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

pre-test loop guard

loop control variable

zero or more iterations

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

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;
```

pre-test loop guard

loop iteration

inverted test

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

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

pre-test loop guard

loop iteration

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

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 */
}
```

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**

Pre loop setup → `mov r1, 0`

Pre-test loop guard → `cmp r1, 10` / `bge .Lendfr`

Post loop update → `add r1, r1, 1`

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

Alternative

Pre-loop setup → `mov r1, 0`

Pre-test loop guard → `cmp r1, 10` / `bge .Lendfr`

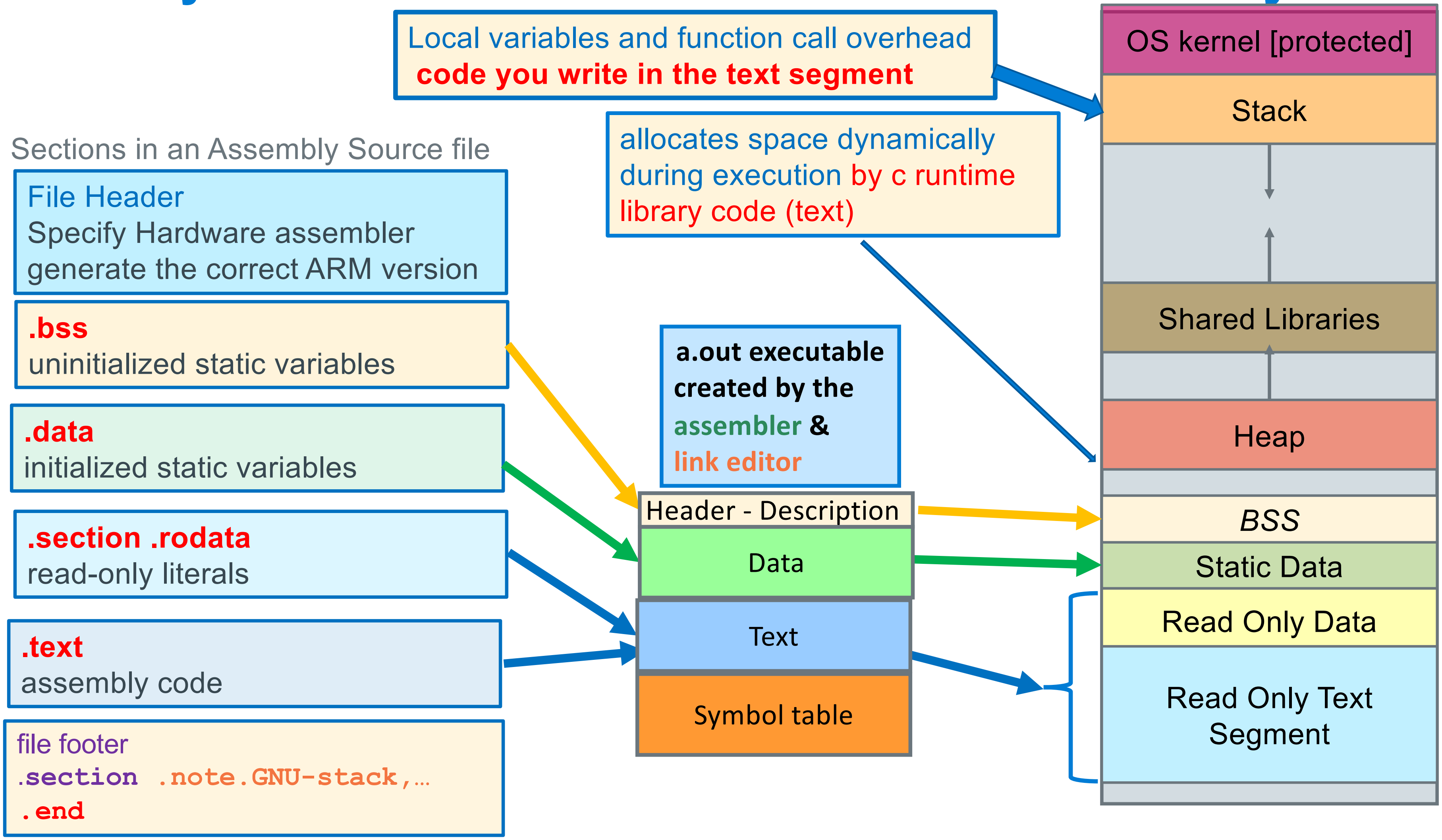
Post loop update → `add r1, r1, 1`

Post-test loop guard → `cmp r1, 10` / `blt .Ldo`

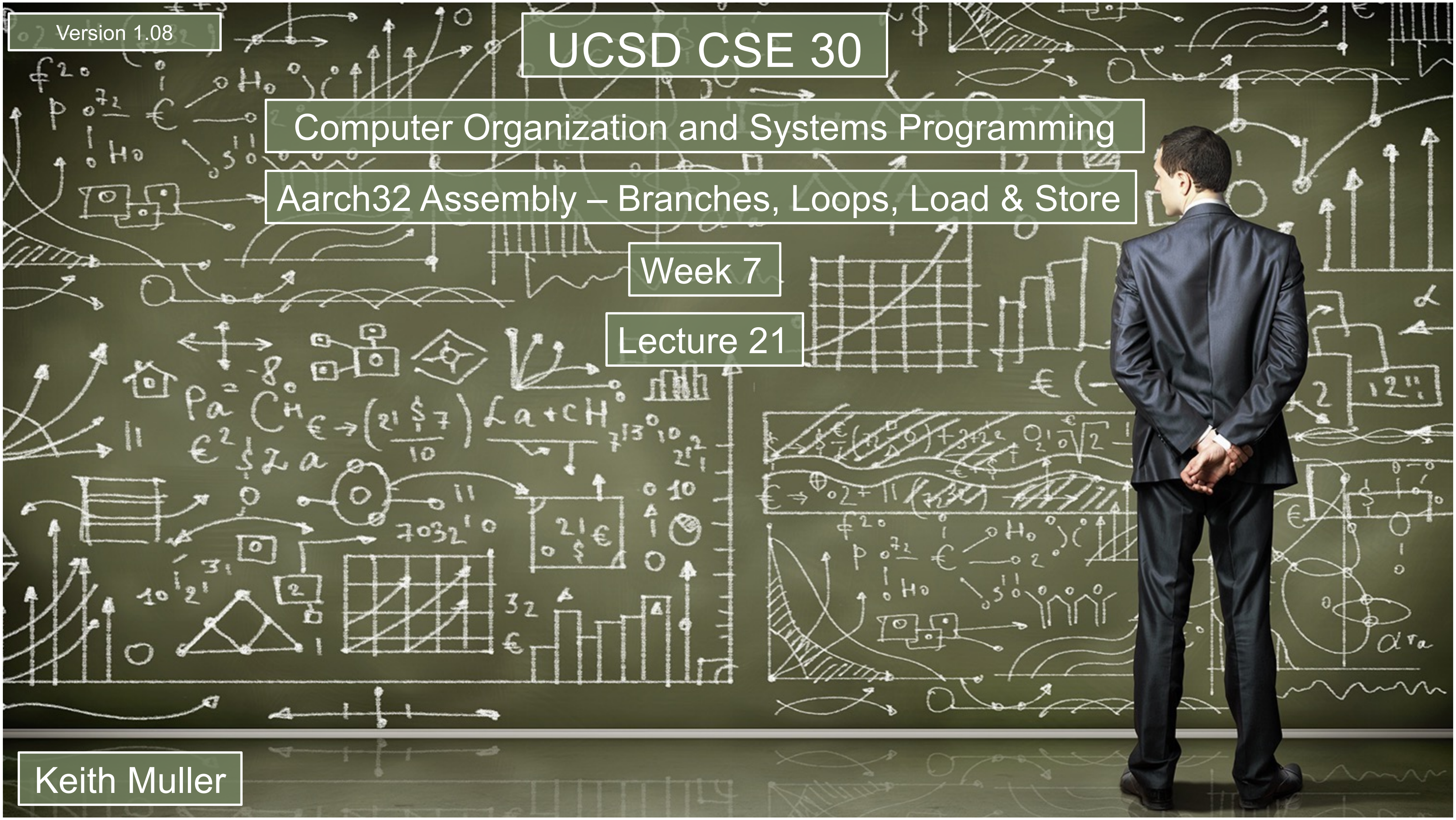
```
    mov r1, 0
    cmp r1, 10
    bge .Lendfr
.Ldo:
    // block code
    add r1, r1, 1
    cmp r1, 10
    blt .Ldo
.Lendfr:
```


x

Assembly Source File to Executable to Linux Memory



x


Version 1.08
UCSD CSE 30
Computer Organization and Systems Programming
Aarch32 Assembly – Branches, Loops, Load & Store
Week 7
Lecture 21
Keith Muller

Nested loops

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

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

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

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

```
    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
```


x

Creating Segments, Definitions In Assembly Source

- The following assembler directives indicate the ***start*** of a **memory segment specification**
  - **Remains in effect** until the next segment directive is seen

```
.bss
        // start uninitialized static segment variables definitions
        // does not consume any space in the executable file
.data
        // start initialized static segment variables definitions
.section .rodata
        // start read-only data segment variables definitions
.text
        // start read-only text segment (code)
```

- Define a literal, static variable or global variable in a segment

```
Label:   .size_directive expression, … expression
```

  - Label: this is the **variables *name***
  - Size_Directive tells the assembler *how much space to* ***allocate*** for that **variable**
- Each **optional** `expression` specifies the contents of one memory location of `.size_directive`
  - `expression` can be in decimal, hex (0x…), octal (0…), binary (0b…), ASCII (' '), string " "

x

Assembly Source File

```
// File Header
        .arch armv6                // armv6 architecture instructions
        .arm                       // arm 32-bit instruction set
        .fpu vfp                   // floating point co-processor
        .syntax unified            // modern syntax

// BSS Segment (only when needed)
        .bss
// Data Segment (only when needed)
        .data
// Read-Only Data (only when needed)
        .section .rodata
// Text Segment – your code
        .text

// Function Header
        .type   main, %function    // define main to be a function
        .global main               // export function name
main:
// function prologue               // stack frame setup
                   // your code for this function here
// function epilogue               //stack frame teardown

// function footer
         .size  main, (. - main)

// File Footer
          .section .note.GNU-stack,"",%progbits // stack/data non-exec
.end
```

- assembly programs end in .S
  - example: test.S
- Always use gcc to assemble
  - _start() and C runtime
- File has a complete program

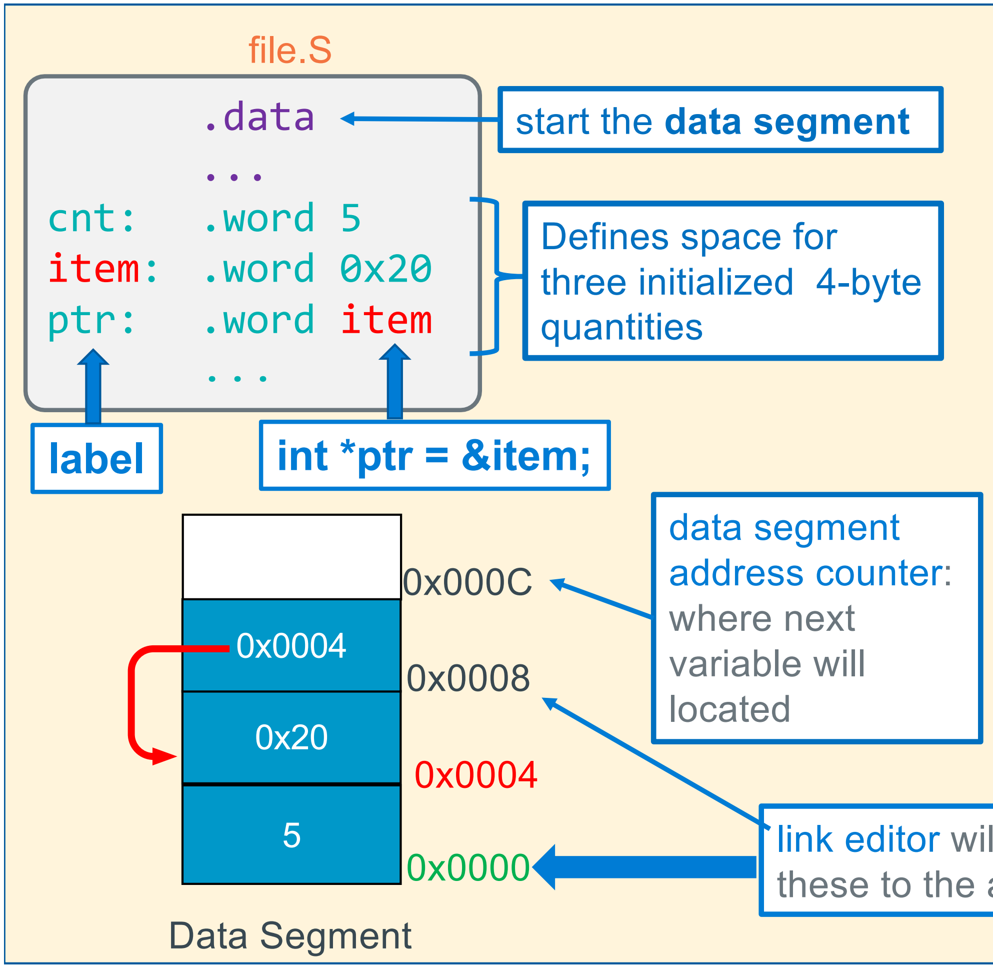
  `gcc file.S`
- File has a partial program

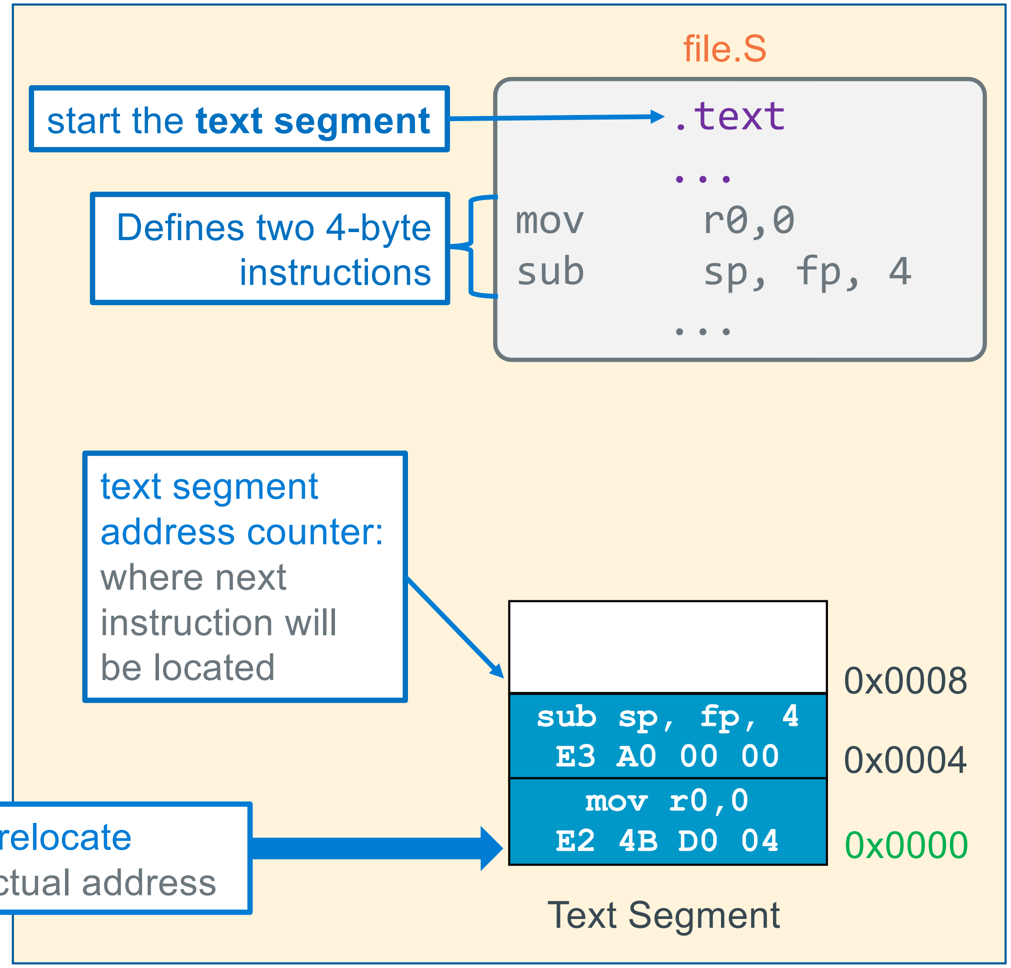
  `gcc -c file.S`
- Link files together

  `gcc file.o cprog.o`

x

Creating A Segment with Assembler Directives





link editor will relocate these to the actual address

x

Defining Static Variables: Allocation and Initialization

Variable SIZE	Directive	Align	C static variable Definition	Assembler static variable Definition
8-bit char (1 byte)	.byte		`char chx = 'A'` `char string[] = {'A','B','C', 0};`	`chx:      .byte 'A'` `string:   .byte 'A','B',0x42,0`
16-bit int (2 bytes)	.hword .short	1	`short length = 0x55aa;`	`length:   .hword 0x55aa`

32-bit int
(4 bytes)
	.word
	.long
		2	`int dist = 5;`
			`int *distptr = &dist;`
			
			`int array[] = {12,~0x1,0xCD,-1};`
				`dist:     .word 5`
				`distptr:  .word dist`
				
				`array:    .word 12,~0x1,0xCD,-3`

strings '\0' term	.string		`char class[] = "cse30";`	`class:    .string "cse30"`

```
int num;
int *ptr = &num;
char msg[] = "123";
char *lit = "456";
```

```
.bss
num:        .word 0
.data
ptr:        .word num
msg:        .string "123"
lit:        .word .Lmsg
.section .rodata
.Lmsg:      .string "456"
```


X

Defining Array Static Variables

```
Label:   .size_directive expression, … expression
```

```
 In C:      int int_buf[100];
            int array[] = {1, 2, 3, 4, 5};
            char buffer[100];
.bss
int_buf:    .space 400   // convert 100 to 400 bytes
char_buf:   .space 100
.data
array:      .word 1, 2, 3, 4, 5
one_buf:    .space 100, 1  // 100 bytes each byte filled with 1
```

`.space size, fill`

- Allocates `size` bytes, each of which contain the value `fill`
- Both `size` and `fill` are absolute expressions
- If the comma and `fill` are **omitted**, `fill` is assumed to be **zero**
- **`.bss section:`** Must be used **without a specified fill**

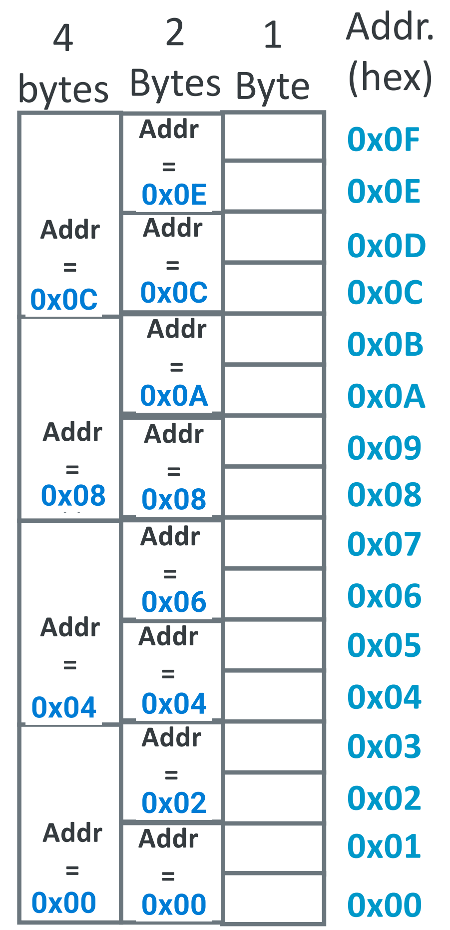
x

Static Variable Alignment: Using .align

Accessing **address aligned** memory based on data type has the best performance

integer: 4 bytes | short: 2 bytes | char: 1

SIZE	Directive	Address ends in	Align Directive
`8-bit char -1 byte`	`.byte`	`0x..0 or 0x..1`	
`16-bit int -2 bytes`	`.hword` `.short`	`0x..0`	`.align 1`
`32-bit int -4 bytes`	`.word` `.long`	`0x..00`	`.align 2`

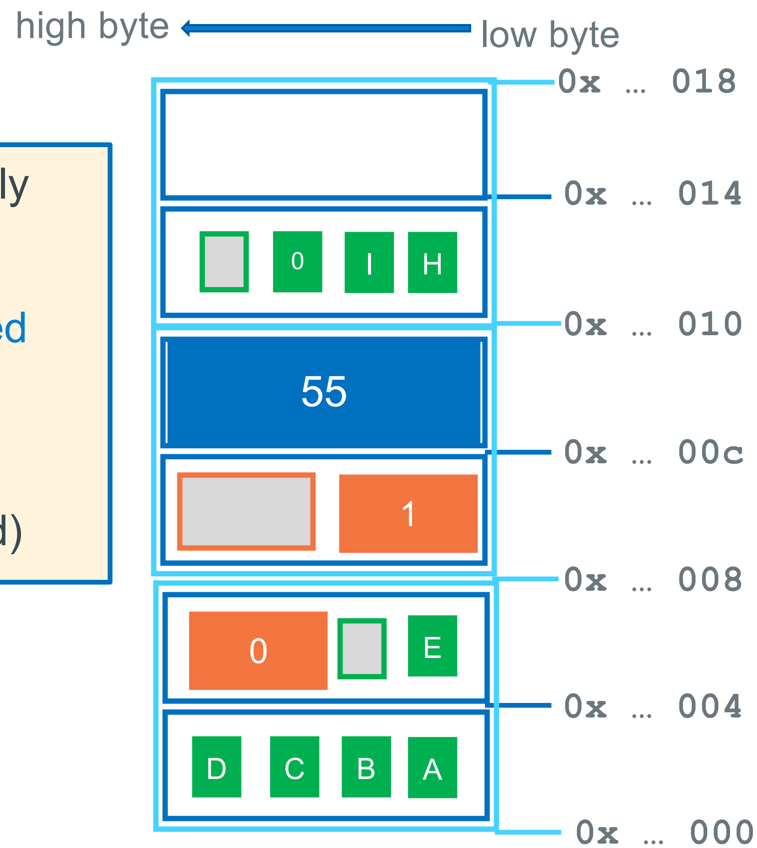


`.align n` **before** variable definition to specify memory alignment requirements

- Tells the assembler **the next line that allocates memory** must **start** at the next higher memory address **where** the lower **n** address bits are zero
- Assembler may adjust the starting address of the **next variable if needed**
- At the **first use of any Segment directive**, alignment **starts at an 8-byte aligned address** (for doubles)

x

Static Variable Alignment Example



`.align n` use is **for the next address allocated** immediately following the `.align` directive

1. Examine both the size and alignment of the variable defined above; determine the address it ends at
2. Add a `.align` if the **next variable** would not be properly aligned based on its type (it is ok if the .align is **not** needed)

```
ch:     .byte 'A','B','C','D','E'
        .align 1   // next 2 bytes
ary:    .hword 0, 1
        .align 2   //next 4 bytes
x:      .word 55

str:    .string "HI"
```


x

Load/Store: Register Base Addressing

`ldr r0, [r1]`

32-bit memory

register r1 (address)

Copies a 32-bit word from the memory location whose address is contained in r1 (r1 is a pointer) into register r0

register r0

r1 is being used as a **pointer** to a location in memory

ldr requires the use of a pointer operand

`str  r0, [r1]`

register r0

Copies all 32 bits of the value held in register r0 to the 32-bit memory location contained in register r1 (r1 pointer)

r1 is being used as a **pointer** to a location in memory

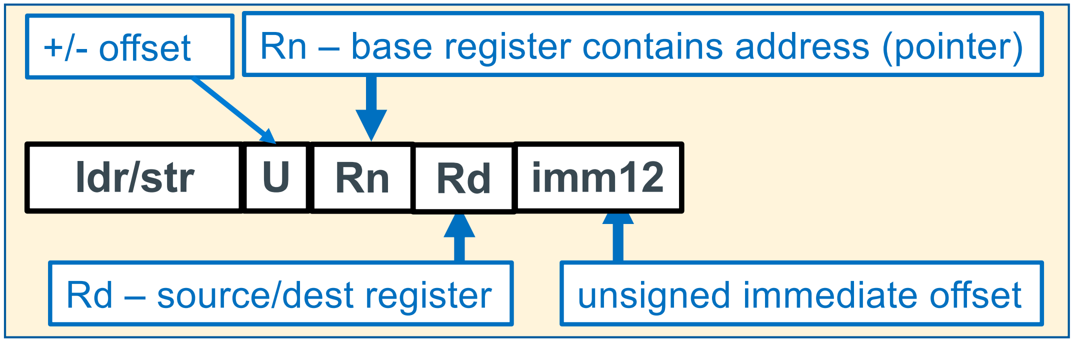
str requires the use of a pointer operand

32-bit memory

register r1 (address)

x

LDR/STR – Base Register + Immediate Offset Addressing

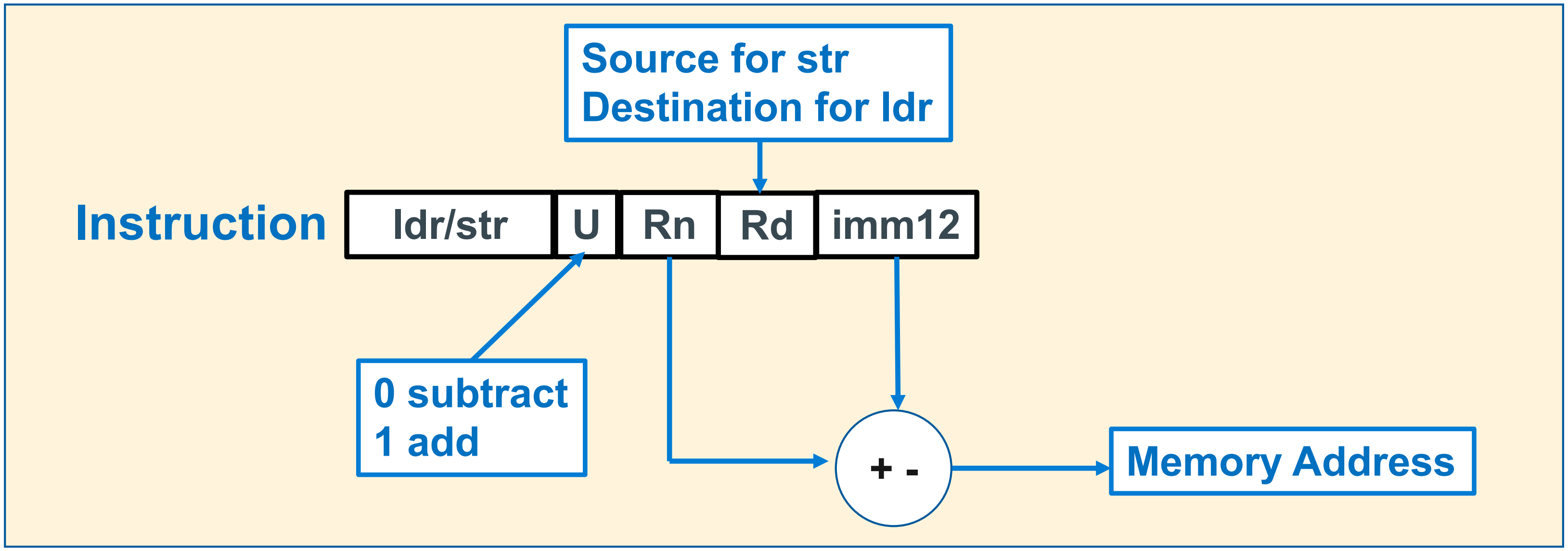


- **Register Base Addressing**:
  - Pointer Address: Rn; source/destination data: Rd
  - **Unsigned pointer address** in stored in the base register
- **Register Base + immediate offset Addressing:**
  - Pointer Address = register content + immediate offset
  - Unsigned offset integer immediate value (bytes) is added or subtracted (U bit above says to add or subtract) from the pointer address in the base register

```
ldr/str  Rd,  [Rn, +- imm12] // base register pointer + offset  imm12 in bytes
                             -4095 <= imm12 <= 4095 (bytes)
ldr/str  Rd,  [Rn]           // base register pointer + 0 offset (imm12 is 0)
```


x

ldr/str Register Base and Register + Immediate Offset Addressing

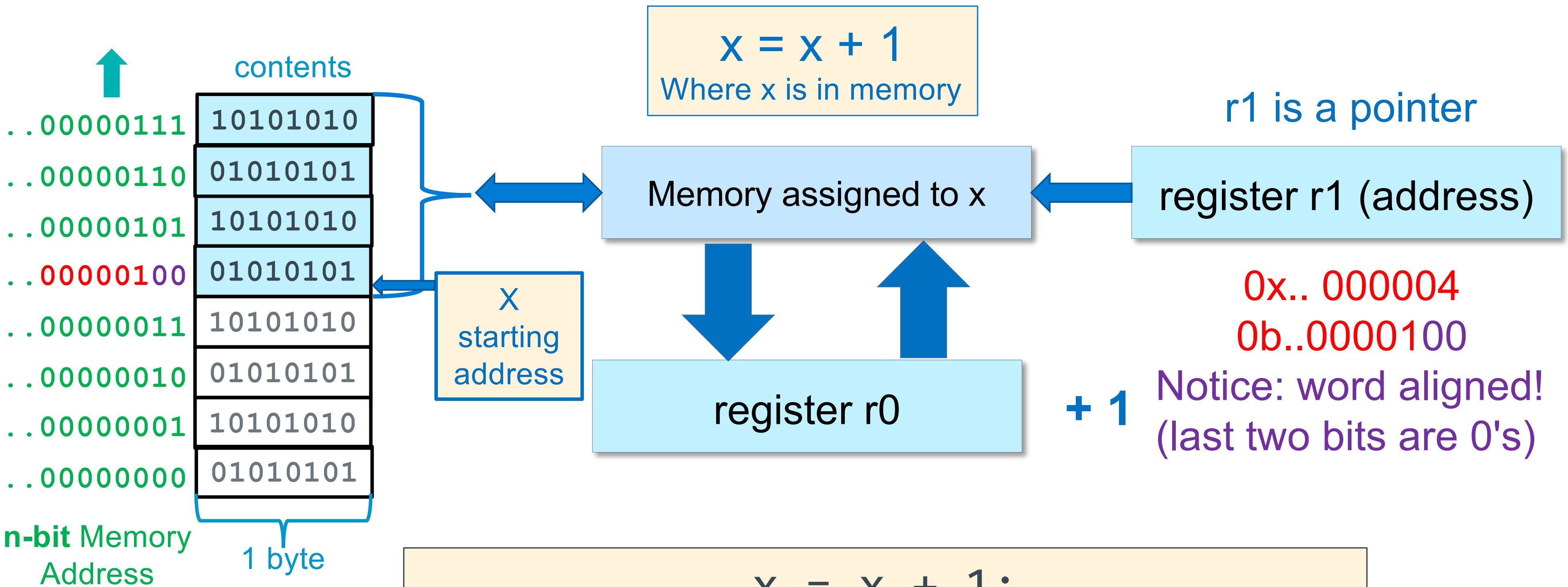


Syntax	Address	Examples

`ldr/str Rd, [Rn +/- constant]`
constant is in bytes
	Rn + or – constant
	same
		`ldr r0, [r5,100]`
		`str r1, [r5, 0]`
		`str r1, [r5]`

x

Example Base Register Addressing Load – Modify – Store




```
x = x + 1;
ldr r0, [r1]          // r0 = *r1 (read x)
add r0, r0, 1         // r0 = r0 + 1 (x++)
str r0, [r1]          // *r1 = r0 write x
```


x

How to Access variables in a Data Segment

```
ldr/str  Rd,  [Rn, +- imm12]
```

- How do you get the **address into the base register Rn** for a Labeled location in memory?
- Assembler **creates a table of pointers** in the **text segment** called the **literal table**
  - It is accessed using **the pc as the base register**
  - Each entry contains a **32-bit Label address**
- How to access this table to get a pointer:

```
ldr/str  Rd, =Label // Rd = address
```

to **load** a **memory** variable
1. load the pointer
2. read (load) from the pointer

to **store** to a **memory** variable
1. load the pointer
2. write (store) to the pointer

assembly source file ex.S

```
        .bss
y:      .space 4
        .data
x:      .word 200
        .section .rodata
.Lmsg:  .string "Hello World"
        .text
        // function header
main:

        // load the contents into r2
        ldr r2, =x      // int *r2 = &x
        ldr r2, [r2]    // r2 = *r2;
        // &x was only needed once above

        // store the contents of r0
        ldr r1, =y      // r1 = &y
        str r0, [r1]    // y = r0
        // keeping &y in r1 above
…
```


x

Extra Slides

ARM Assembly Source File: Header

File Header
At the top of every ARM source file

```
        .arch   armv6        // armv6 architecture
        .arm                 // arm 32-bit instruction set
        .fpu    vfp          // floating point co-processor
        .syntax unified      // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

`.arch <architecture>`

- Specifies the target architecture to generate machine code
- Typically specify oldest ARM arch you want the code to run on – most arm CPUs are backwards compatible

`.arm`

- Use the 32-bit ARM instructions, There is an alternative 16-bit instruction set called thumb that we will not be using

`.fpu <version>`

- Specify which floating point co-processor instructions to use (OPTIONAL we will not be using floating point)

X

ARM Assembly Source File: Header and Footer

File Header
At the top of every ARM source file

```
        .arch   armv6          // armv6 architecture
        .arm                   // arm 32-bit instruction set
        .fpu    vfp            // floating point co-processor
        .syntax unified        // modern syntax
```

```
// Contents of the other memory segment include .text (your code)
```

File Footer
At the bottom of every ARM source file

```
.section .note.GNU-stack,"",%progbits // set stack/data non-exec
.end
       // everything past the .end is ignored!
       // Debugging notes etc
```

`.syntax unified`

- use the standard ARM assembly language syntax called ***Unified Assembler Language* (*UAL*)**

`.section .note.GNU-stack,"",%progbits`

- tells the linker to **make the stack and all data segments not-executable** (no instructions in those sections) – security measure

`.end`

- at the end of the source file, everything written after the .end is ignored

x

Checking alignments

```
        .data                // al1.s
ch:     .byte 'A','B','C','D','E'
ary:    .hword 0, 1
x:      .word 55
str:    .string "HI"
dd:     .double 8.1
```

```
% gcc -c -Wa,-ahlns al1.s
   1                     .data
   2 0000 41424344 ch: .byte 'A','B','C','D','E'
   2      45
   3 0005 00000100 ary: .hword 0, 1
   4 0009 37000000 x: .word 55
   5 000d 484900   str: .string "HI"
   6 0010 33333333 dd: .double 8.1
   6      33332040
```

address contents

- pass args to gas with –Wa,-<gas_args>
- Use: `%gcc -c -Wa,-ahlns al1.s`

```
        .data                // al1.s
ch:     .byte 'A','B','C','D','E'
        .align 1
ary:    .hword 0, 1
        .align 2
x:      .word 55
str:    .string "HI"
        .align 3
dd:     .double 8.1
```

```
gcc -c -Wa,-ahlns al1.s
   1                     .data
   2 0000 41424344 ch: .byte 'A','B','C','D','E'
   2      45
   3 0005 00            .align 1
   4 0006 00000100 ary: .hword 0, 1
   5 000a 0000          .align 2
   6 000c 37000000 x: .word 55
   7 0010 484900   str: .string "HI"
   8 0013 00000000 .align 3
   8      00
   9 0018 33333333 dd: .double 8.1
   9      33332040
```

X

Function Header and Footer Assembler Directives

function entry point
address of the first
instruction in the function
Must not be a local label
(does not start with .L)

```
                .text
Function        .global myfunc              // make myfunc global for linking
  Header        .type   myfunc, %function  // define myfunc to be a function
                .equ    FP_OFF,   4         // fp offset in main stack frame
myfunc:
                // function prologue, stack frame setup
                // your code
                // function epilogue, stack frame teardown
Function        .size myfunc, (. - myfunc)
  Footer
```

`.global function_name`

- Exports the function name to other files. **Required for main function,** optional for others

`.type name, %function`

- The `.type` directive sets the **type of a symbol/label name**
- `%function` specifies that **name** is a function (name is the address of the first instruction)

`equ FP_OFF, 4`

- Used for basic stack frame setup; the number 4 will change – later slides

`.size name, bytes`

- The `.size` directive is used to set the size associated with a symbol
- Used by the linker to exclude unneeded code and/or data when creating an executable file
- It is also used by the **debugger** gdb
- `bytes` **is best calculated as an expression: (period is the current address in a memory segment)**

  In CSE30 required use: `.size name, (. - name)`

X