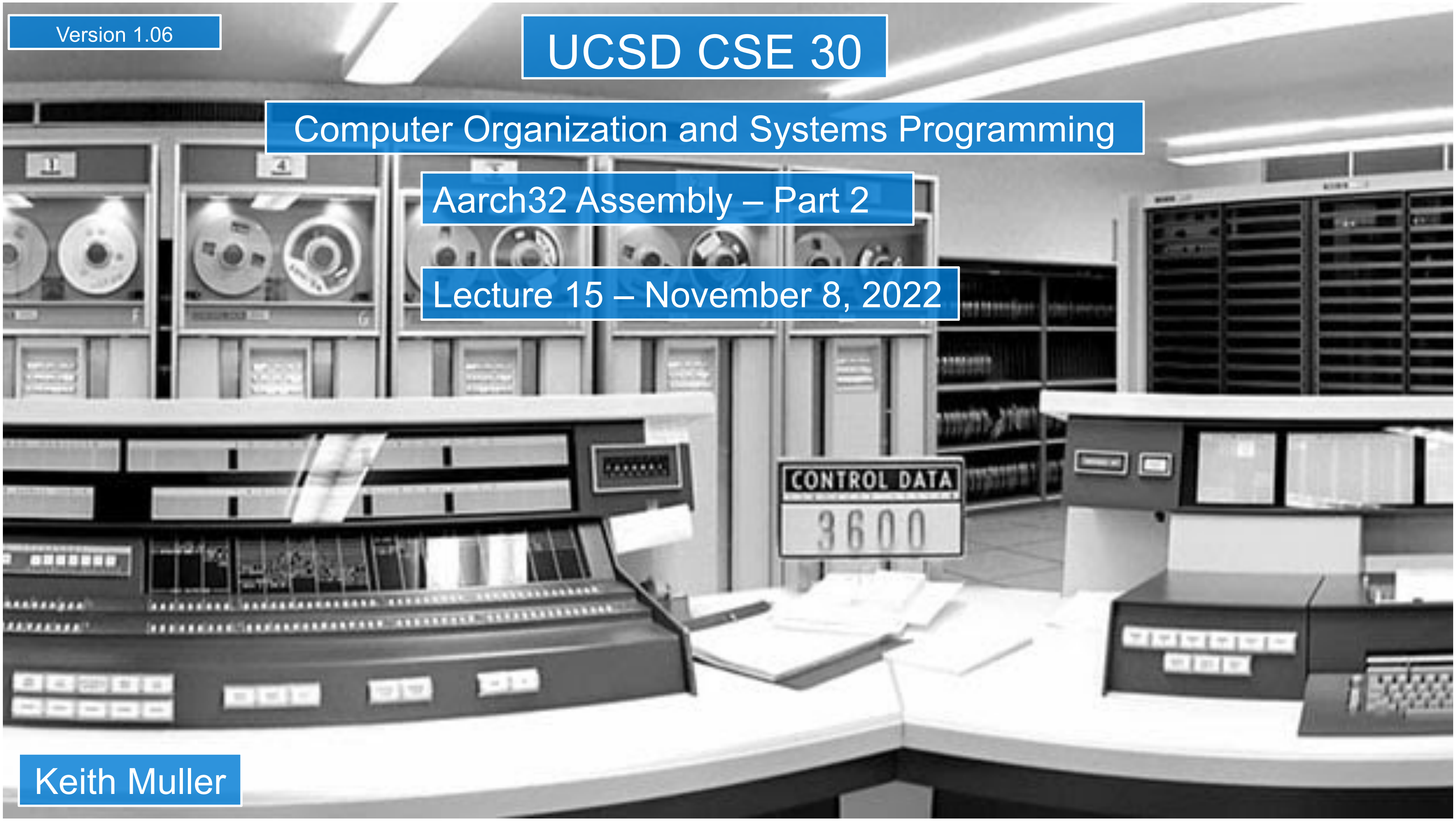
Version 1.06

UCSD CSE 30

Computer Organization and Systems Programming

Aarch32 Assembly – Part 2

Lecture 15 – November 8, 2022



Keith Muller

Conditional Branch: Changing the Next Instruction to Execute

```
        cmp     r3, r4       <- Branch guard
        beq    .Ldone
        // otherwise fall through do the add
        add    r1, r2, 1
        // now fall through to mov
.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
```

How to implement a **branch/loop guard in CSE30**

1. Use a **cmp/cmm** instruction to set the condition bits
2. Follow the **cmp/cmm with one or more variants of the <u>conditional</u> branch instruction**
   - **Conditional branch instructions** if evaluate to true (based on the flags set by the cmp) the next instruction will the one at the branch label
   - Otherwise, execution falls through to the instruction that immediately follows the branch

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

x

Examples: Guards (Conditional Tests) and their Inverse

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

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

Eliminate unnecessary branches and labels: use Fall Throughs

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

```
      mov   r2, 0
      add   r1, r2, r3
```

Use fall-through!
do not branch to a Label
on the **next statement**!

x

Branching, What not to do: Spaghetti Code

```
        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 branch** commands (conditional or unconditional) is an indication you should look to reorganize your code

To the left are many unreachable sections of code

Much faster and easier to read!

```
        mov     r1, 1
        mov     r2, 2
        mov     r3, 3
        mov     r4, 4
        mov     r7, 7
```


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;`	`    cmp r0, 5`	`    cmp r0, 5`
`if (r0 == 5) {`	`    beq .Lendif`	`    bne .Lendif`
`    //code`	`    //code`	`    // code`
`}`	`.Lendif:`	`.Lendif:`

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

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

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

If r0 != 5 true
then branch ***around***
the 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 */
    /* then fall through */
}
```

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


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

If statement examples – Branch Around the True block!

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

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

Fall through

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

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

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

Fall through

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

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

Fall through

x

Branching: Using Fall through!

Some call this "goto like" structure

- Do not use unnecessary branches 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
```

Caution!
Two adjacent branches

Do the following:

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


x

Anatomy of a Conditional Branch: If - Else statement

Branch condition Test (branch guard)

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

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 */
        /* fall through */
}
```

condition true block

condition false block

x

Program Flow: If with an Else

```
if (r0 == 5) {
    /* cond. true block */
    /* branch around false */
} 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;
    // now branch around else
} else {
    r2 = 0;
    /* fall through */
}
r4 = r2 + r4;
```

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

Same test
swapped blocks

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

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

```
    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

Bad Style: Branching Upwards (When 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
        blt .Lendif
        beq .Lelse1
        bgt .Lelse2
.Lelse1:
        mov   r2, 0
        b .Lendif
.Lelse2:
        add r4, r3, r2
        b .Lelse1
.Lendif:
        add   r4, r2, r4
```

Upwards branch

eliminate this branch replace with fall through

eliminate this branch

```
        cmp r3, r4
        blt .Lendif
        beq .Lelse1
        add r4, r3, r2
.Lelse1:
        mov   r2, 0
.Lendif:
        add   r4, r2, r4
```


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 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

```c
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 – 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 the 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

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

if stmt 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
```


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        ->     mov r1, 0
                          .Lfor:
Pre-test loop guard   ->      cmp r1, 10
                              bge .Lendfr
                              // block code
Post loop update      ->      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        ->     mov r1, 0
Pre-test loop guard   ->     cmp r1, 10
                             bge .Lendfr
                         .Ldo:
                             // block code
Post loop update      ->     add r1, r1, 1
Post-test loop guard  ->     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;
```

- 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

```
    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

Keep loops Properly Nested:
Do not branch into the middle of a loop

- It is hard to understand and debug loops when you branch into the middle of a loop
- **Keep loops proper nested**

Bad practice: branch into loop body

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

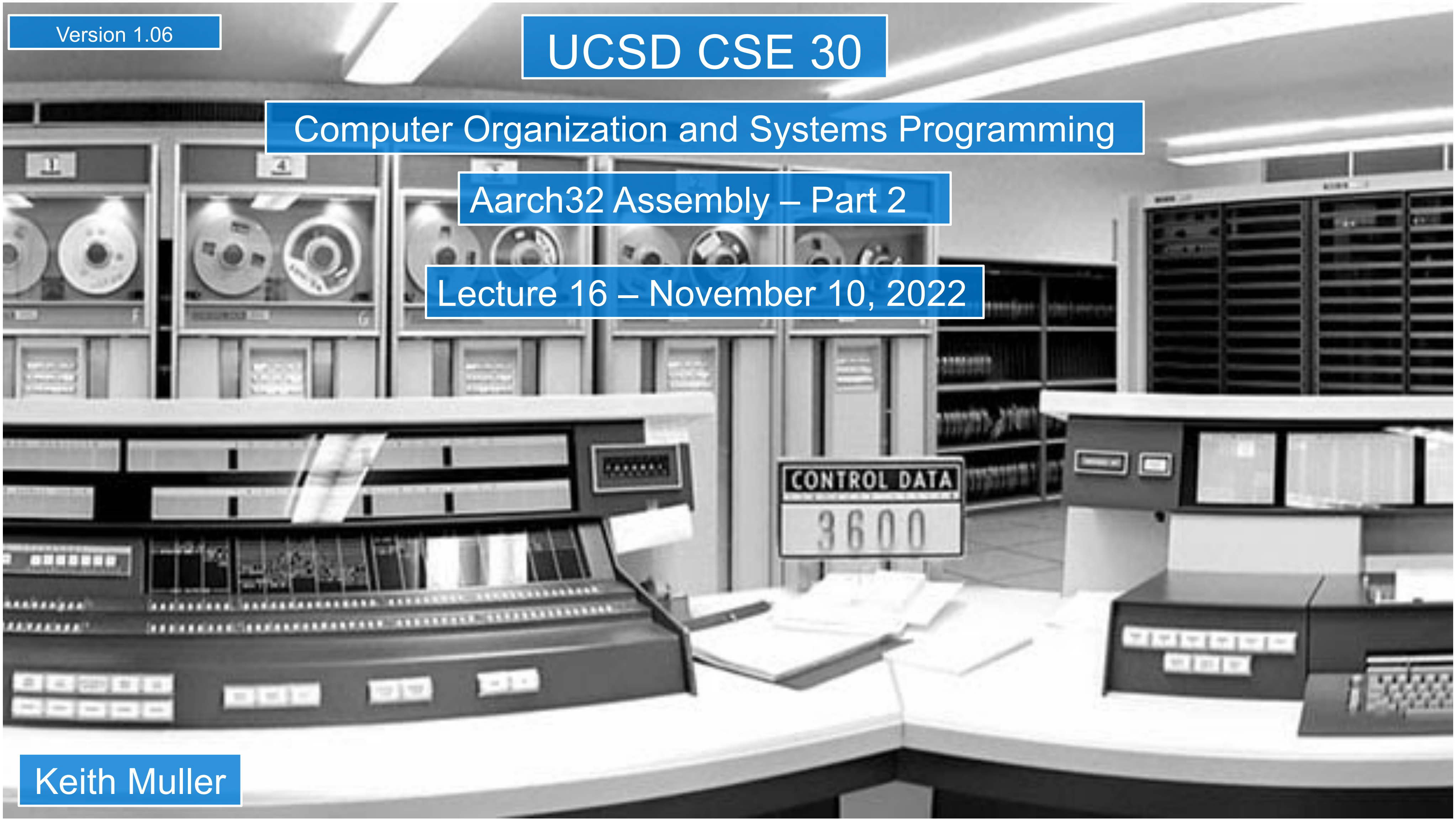
Version 1.06

UCSD CSE 30

Computer Organization and Systems Programming

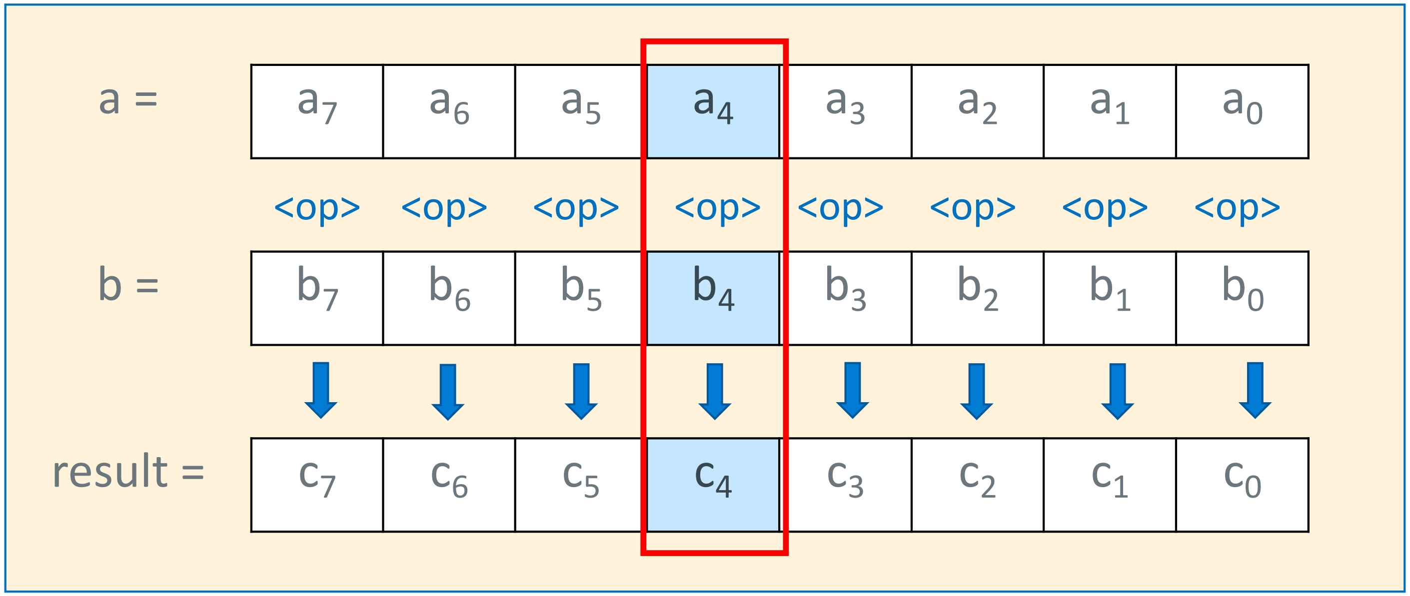
Aarch32 Assembly – Part 2

Lecture 16 – November 10, 2022



Keith Muller

What is a Bitwise Operation?



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

x

Bitwise (Bit to Bit) Operators in C

```
output = ~a;
```

a	~a
0	1
1	0

```
output = a & b;
```

a	b	a & b
0	0	0
0	1	0
1	0	0
1	1	1

& with 1 to let a **bit through**
& with 0 to **set a bit to 0**

```
output = a | b;
```

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

| with 1 to **set a bit to 1**
| with 0 to let a **bit through**

```
output = a ^ b; //EOR
```

a	b	a ^ b
0	0	0
0	1	1
1	0	1
1	1	0

^ with 1 **will flip the bit**
^ with 0 to let a **bit through**

Bitwise NOT

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

Bitwise AND

```
  0110
& 1100
  ----
  0100
```

Bitwise OR

```
  0110
| 1100
  ----
  1110
```

Bitwise EOR

```
  0110
^ 1100
  ----
  1010
```


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 (not)

```
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

```
mvn   r0, 12

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

0x0c

register r0

0xffff fff3

Bitwise NOT

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

- A **bitwise NOT** operation

0x 0c

imm8 expansion

0x0000000c

bitwise not

0xfffffff3

x

mvn – Copies NOT (~) (1's Compliment Copy)

mvn	Rd	rot4	imm8

Rd: destination register; rot4, imm8: constant (immediate value)

mvn	Rd	Rm

Rd: destination register; Rm: source register

```
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
```

invert the bits — copy into 32 bits zero extend

r1 0xfffffffb ← 0x00000004 ← 0x4

r1 0x55555555 ← 0xaaaaaaaa r5

r1 0x11111111 ← 0x0

x

Bitwise versus C Boolean Operators

Meaning	*Operator*	*Operator*	*Meaning*
`Boolean AND`	`a && b`	`a & b`	`Bitwise AND`
`Boolean OR`	`a || b`	`a	b`	`Bitwise OR`
`Boolean NOT`	`!b`	`~b`	`Bitwise NOT`

Boolean operators **act on the entire value not the individual bits**

& versus &&

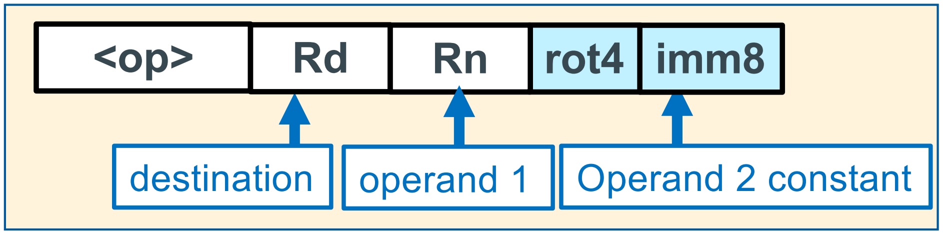
```
0x10 &   0x01 = 0x00 (bitwise)
0x10 &&  0x01 = 0x01 (Boolean)
```

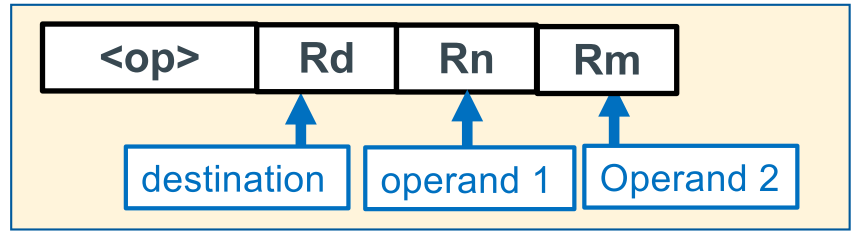
`! versus ~`

```
~0x01 = 0xfffffffe (bitwise)
!0x01 = 0x0 (Booelan)
```


x

Bitwise Instructions






```
<op>  Rd,  Rn, constant   // Rd = Rn <op> constant
<op>  Rd,  constant       // Rd = Rd <op> constant
<op>  Rd,  Rn,  Rm        // Rd = Rn <op> Rm
```

Bytes: 0 <= imm8 <= 255 + values from "rotating" rot 4 bits

Bitwise <op> description	C Syntax	***Arm <op> Syntax Op2: either register or constant value***	***Operation***
Bitwise **AND**	a & b	and R_d, R_n, Op2	R_d = R_n & Op2
Bit Clear each bit in Op2 that is a 1, the same bit in R_d, is cleared	a & ~b	bic R_d, R_n, Op2	R_d = R_n & ~Op2
Bitwise **OR**	a | b	orr R_d, R_n, Op2	R_d = R_n | Op2
Exclusive **OR**	a ^ b	eor R_d, R_n, Op2	R_d = R_n ^ Op2

x

The act (operation) of *Masking*

a =	a_7	a_6	a_5	a_4	a_3	a_2	a_1	a_0
	<op>	<op>	<op>	<op>	<op>	<op>	<op>	<op>
b =	b_7	b_6	b_5	b_4	b_3	b_2	b_1	b_0
result =	c_7	c_6	c_5	c_4	c_3	c_2	c_1	c_0

- Bit masks access/modify specific bits in memory
- Masking act of applying a mask to a value with a specific op:
- `orr:` 0 passes bit unchanged, 1 sets bit to 1 `(a = b | c; // in C)`
- `eor:` 0 passes bit unchanged, 1 inverts the bit `(a = b ^ c; // in C)`
- `bic:` 0 passes bit unchanged, 1 clears it `(a = b & ~c; // in C)`
- `and:` 0 clears the bit, 1 passes bit unchanged `(a = b & c; // in C)`

x

Mask on and Mask off

force bits to 1 "**mask on**" operation

- 1 to **set a bit to 1**
- 0 to let a **bit through unchanged**

```
orr  r1, r2, r3

r1 = r2 | r3; // in C
```

Example: force lower 16 bits to 1

```
DATA: r2 0xab ab ab 77
orr
MASK: r3 0x00 00 ff ff
```

unchanged   forces to a 1

```
RSLT: r1 0xab ab ff ff
```

Example: force lower 8 bits to 1

```
DATA: r2 0xab ab ab 77

orr   r1   r2, 0xff

r1 = r2 | 0xff; // in C

RSLT: r1 0xab ab ff ff
```


x

Mask on and Mask off

force bits to 0 "**mask off**" operation

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

```
and  r1, r2, r3

r1 = r2 & r3; // in C
```

Example: force lower 8 bits to 0

```
DATA: r2 0xab ab ab 77
and
MASK: r3 0xff ff ff 00
  unchanged           forces to a 0
RSLT: r1 0xab ab ab 00
```

Example: force lower 8 bits to 0

```
DATA: r2 0xab ab ab 77

and r1  r2, 0xffffff00

r1 = r2 & 0xffffff00; // in C

RSLT: r1 0xab ab ab 00
```


x

Mask off versus Bit Clear

force bits to 0 "**mask off**" operation

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

```
and  r1, r2, r3

r1 = r2 & r3; // In C
```

clear specific bits which are 1 in a mask, 0 bits in the mask are unchanged

```
r1 = r2 & ~r3;  // in C

bic  r1, r2, r3
```

Example: force only bit 5 to 0

```
DATA: r2 0xab ab ab 77      77: 0111 0111
and
MASK: r3 0xff ff ff df      df: 1101 1111
     unchanged        forces to a 0
RSLT: r1 0xab ab ab 57      57: 0101 0111

r1 = r2 & 0xffffffdf; // In C
```

clear bit 5 to a 0 without changing the other bits

```
DATA: r2 0xab ab ab 77
bic                          r3: 0010 0000
                            ~r3: 1101 1111
MASK: r3 0x00 00 00 20
     unchanged        clears bit 5
RSLT: r1 0xab ab ab 57

r1 = r2 & ~0x00000020;  // in C
```

x

Extracting (Isolate) a Field of Bits with a mask

extract top 8 bits of r2 into r1

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

```
and  r1, r2, r3
```

```
DATA: r2 0xab ab ab 77
and
MASK: r3 0xff 00 00 00
   unchanged          forces to a 0
RSLT: r1 0xab 00 00 00
```

extract top 8 bits of r2 into r1

```
DATA: r2 0xab ab ab 77

and  r1, r2, 0xff000000

RSLT: r1 0xab 00 00 00

r1 = r2 & 0xff000000;  // in C
```


x

Finding if a bit is set

query the status of a bit "**bit status**" operation

- 0 to **set a bit to 0** ("clears the bit")
- 1 to let a **bit through unchanged**

```
    and  r1, r2, 0x02
    cmp r1, 0
    beq .Lendif
    // code for is set
.Lendif:
```

```
Example is bit 1 set

DATA: r2 0xab ab ab 77

and

MASK:    0x00 00 00 02   is bit 1 set?

forces to a 0              unchanged

RSLT: r1 0x00 00 00 02   != 0 if set
```

```
unsigned int r1, r2;
// code
r1 = r2 & 0x02
if  (r1 != 0) {
    // code for is set
}
```

```
unsigned int r2;
// code
if ((r2 & 0x02) != 0) {
    // code for is set
}
```


x

Even/Odd

Even or odd, check LSB (same as mod %2)

```
check LSB (bit 0) if set then odd, else even

        and  r1, r2, 0x01

        cmp  r1, 0x01

        bne .Lendif

        // code for handling odd numbers

.Lendif:
```

```
DATA: r2 0xab ab ab 77

and

MASK: r3 0x00 00 00 01 (mod 2 even or odd)

forces to a 0             unchanged

RSLT: r1 0x00 00 00 01 (odd)
```

```
unsigned int r2;
// code
if ((r2 & 0x01) != 0) {
    // code for handling odd numbers
}
```

MOD %<power of 2>

remainder (mod): num % d where num ≥ 0 and d = 2^k

mask = 2^k -1 so for mod 16, mask = 16 -1 = 15

```
and  r1, r2, r3
```

```
Example: %2
DATA: r2 0xab ab ab 77
and
MASK: r3 0x00 00 00 01 (mod 2 even or odd)
 forces to a 0            unchanged
RSLT: r1 0x00 00 00 01 (odd)
```

```
Example: Mod 16
DATA: r2 0xab ab ab 77
and
MASK: r3 0x00 00 00 0f (mod 16)
 forces to a 0            unchanged
RSLT: r1 0x00 00 00 07
```

Flipping bits: bit toggle
Used in PA8

invert (*flip*) bits "**bit toggle**" operation

- 1 **will flip the bit**
- 0 to let a **bit through**

```
eor   r1, r2, r3
```

- Observation: When applied twice, it returns the original value (symmetric encoding)
- With a mask of all 1's is a 1's compliment

```
Example: flip the lower 8-bits

    eor  r1, r2, 0xff
```

```
unsigned int r1, r2;

r1 = r2 ^ 0xff;
```

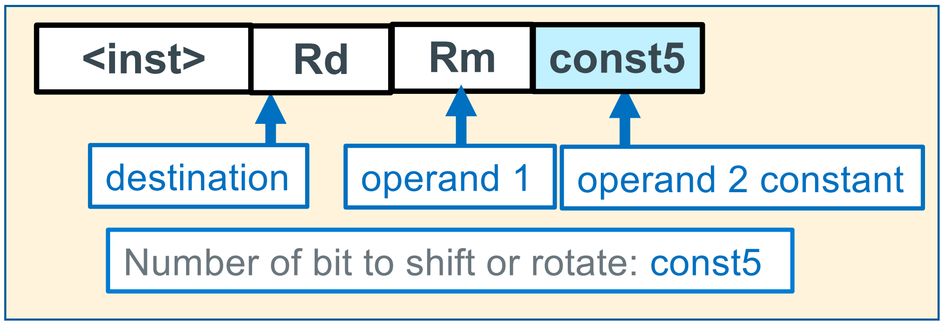
Example: invert (*flip*) the lower 8-bits

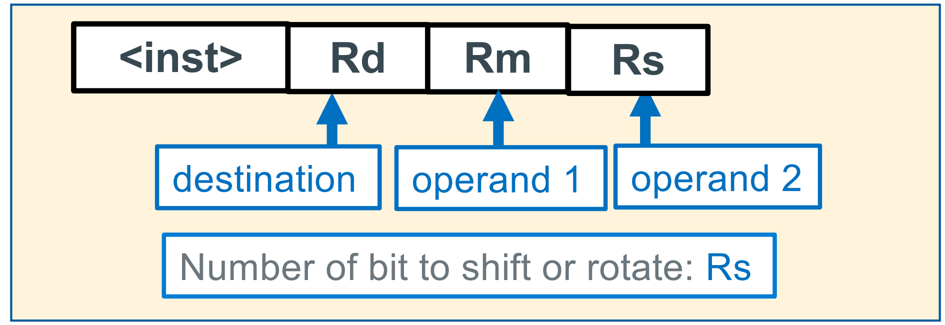
```
DATA: r2 0xab ab ab 77   77: 0111 0111

eor

MASK: r3 0x00 00 00 ff

   unchanged             inverts (flips)

RSLT: r1 0xab ab ab 88   88: 1000 1000
```

```
DATA: r1 0xab ab ab 88

eor

MASK: r3 0x00 00 00 ff apply a 2nd time

                        inverts (flips)

RSLT: r1 0xab ab ab 77 original value!
```


x

Shift and Rotate Instructions



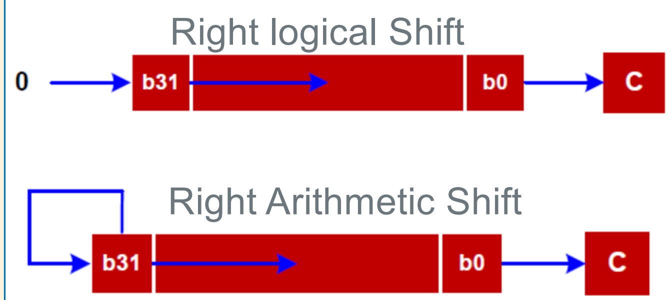


Instruction	*Syntax*	*Operation*	*Notes*	*Diagram*
Logical Shift Left int x; or unsigned int x x << n;	LSL R_d, R_m, *const5* LSL R_d, R_m, R_s	$R_d \leftarrow R_m$ << *const5* $R_d \leftarrow R_m$ << R_s	Zero fills shift: 0 - 31	
Logical Shift Right unsigned int x; x >> n;	LSR R_d, R_m, *const5* LSR R_d, R_m, R_s	$R_d \leftarrow R_m$ >> *const5* $R_d \leftarrow R_m$ >> R_s	Zero fills shift: 1 - 32	
Arithmetic Shift Right int x; x >> n;	ASR R_d, R_m, *const5* ASR R_d, R_m, R_s	$R_d \leftarrow R_m$ >> *const5* $R_d \leftarrow R_m$ >> R_s	Sign extends shift: 1 - 32	
Rotate Right unsigned int x; x = (x>>n)|(x<<(32-n)) ;	ROR R_d, R_m, *const5* ROR R_d, R_m, R_s	$R_d \leftarrow R_m$ ror *const5* $R_d \leftarrow R_m$ ror R_s	right rotate rot: 0 - 31	

x

Shift Operations in C

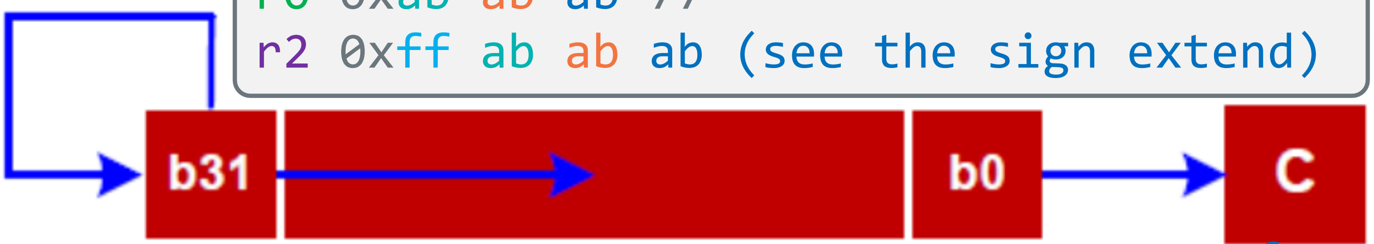
- n is number of bits to shift a variable x of width w bits
- Shifts by `n < 0` or `n ≥ w` are *undefined*
- Left shift (`x << N`) – **Multiplies by 2^N**
  - Shift N bits left, Fill with `0`s on right
- **In C:** behavior of >> is determined by compiler
  - gcc: it depends on data type of `x` (signed/unsigned)
- Right shift (`x >> N`) - **Divides by 2^N**
  - Logical shift (for unsigned variables)
    - Shift N bits right, Fill with 0s on left
  - Arithmetic shift (for signed variables) – Sign Extension
    - Shift N bits right while **<u>Replicating</u>** the most significant bit on left
    - Maintains sign of `x`
- **In Java:** logical shift is **>>>** and arithmetic shift is >>



X

Arithmetic Shift Right (there is no arithmetic shift left)

```
asr r2, r0, 8

r0 0xab ab ab 77
r2 0xff ab ab ab (see the sign extend)
```



Test for sign
-1 if r0 negative

```
  asr r2, r0, 31
  cmp r2, -1
  bne .Lendif
  //code neg #
.Lendif:
```

```
r0 0xab ab ab 77
r2 0xff ff ff ff
```

```
int i;
//code
if ((i>>31) == -1) {
  // code neg #
}
```

```
int i;
//code
if ((i>>31) == 0) {
  // code pos #
}
```

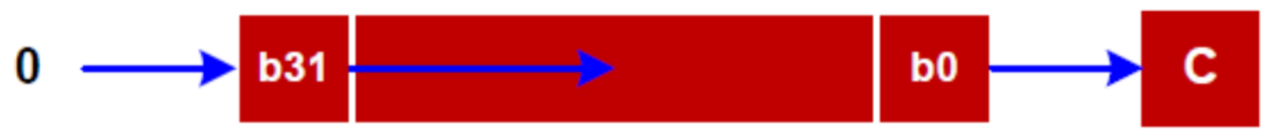
Test for sign
0 if r0 positive

```
  asr r2, r0, 31
  cmp r2, 0
  bne .Lendif
  //code positive #
.Lendif:
```

```
r0 0x7b ab ab 77
r2 0x00 00 00 00
```


x

Logical Shift & Rotate Operations




```
lsr r2, r0, 8

r0 0xab ab ab 77
r2 0x00 ab ab ab
```


```
lsl r2, r0, 8

r0 0xab ab ab 77
r2 0xab ab 77 00
```

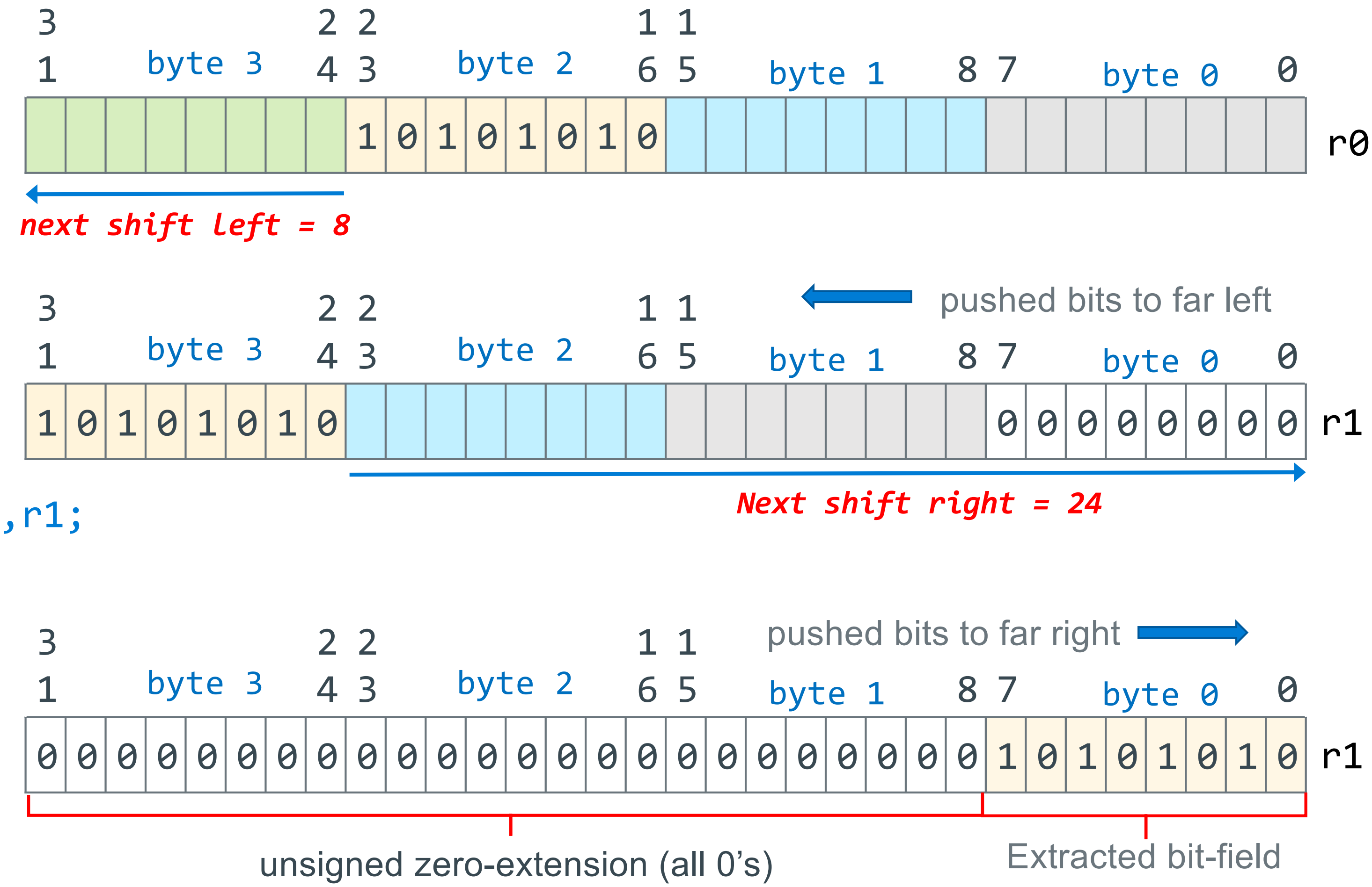

```
ror r2, r0, 8

r0 0xab ab ab 77
r2 0x77 ab ab ab
```


x

Extracting/Isolating Unsigned Bitfields

Hint: Useful for PA8

- Move byte 2 in r0 to byte 0 in r1




```
lsl   r1, r0, 8
```

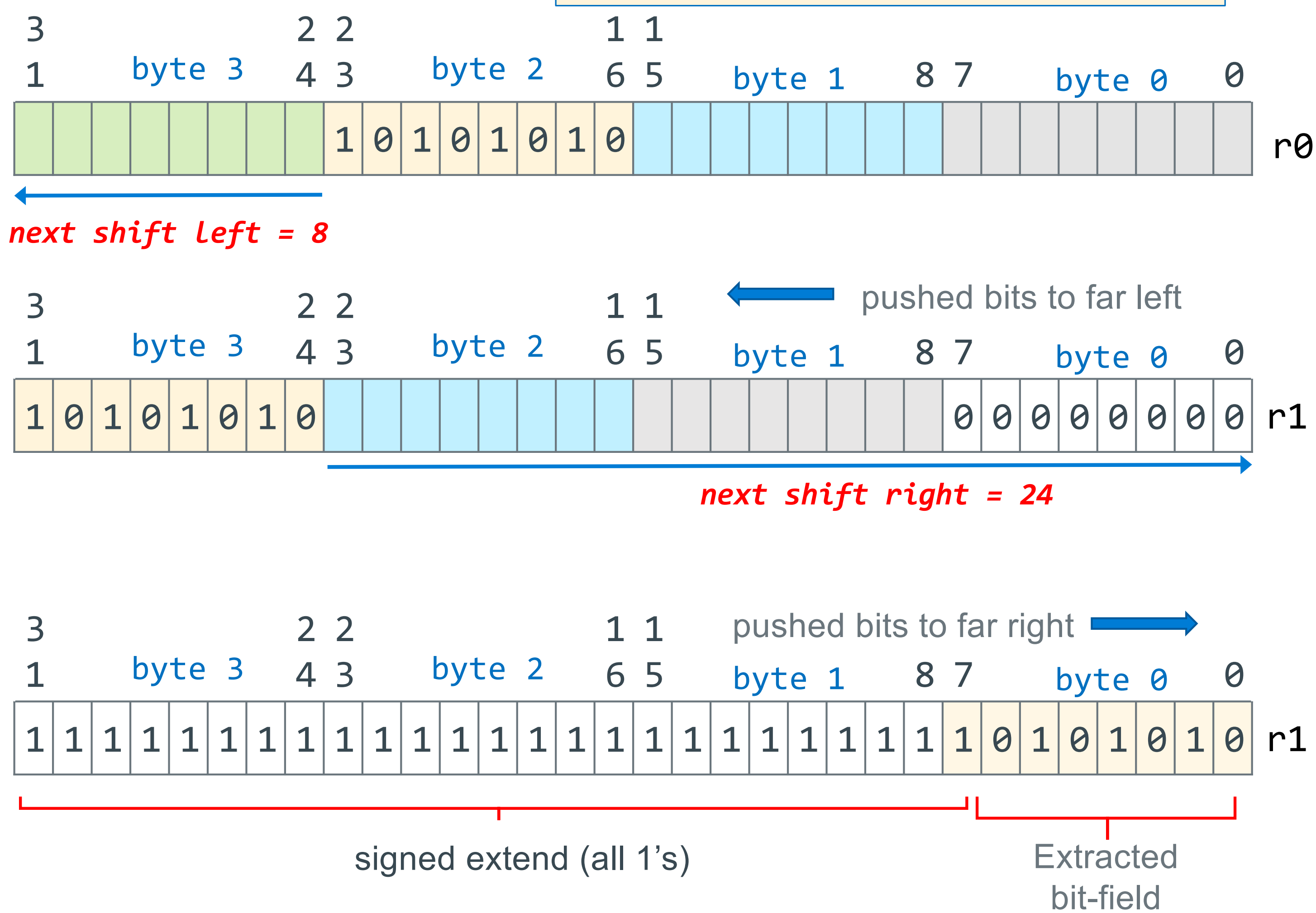
```
unsigned int r0,r1;
r1 = r0 << 8;
```

```
lsr   r1, r1, 24
r1 = r1 >> 24;
```


x

Extracting Signed Bitfields

- Move byte 2 in r0 to byte 0 in r1




```
lsl  r1, r0, 8
int r0,r1;
r1 = r0 << 8;
```

```
asr  r1, r1, 24
r1 = r1 >> 24;
```


x

Inserting Bitfields – Inserting Source Field into Destination Field

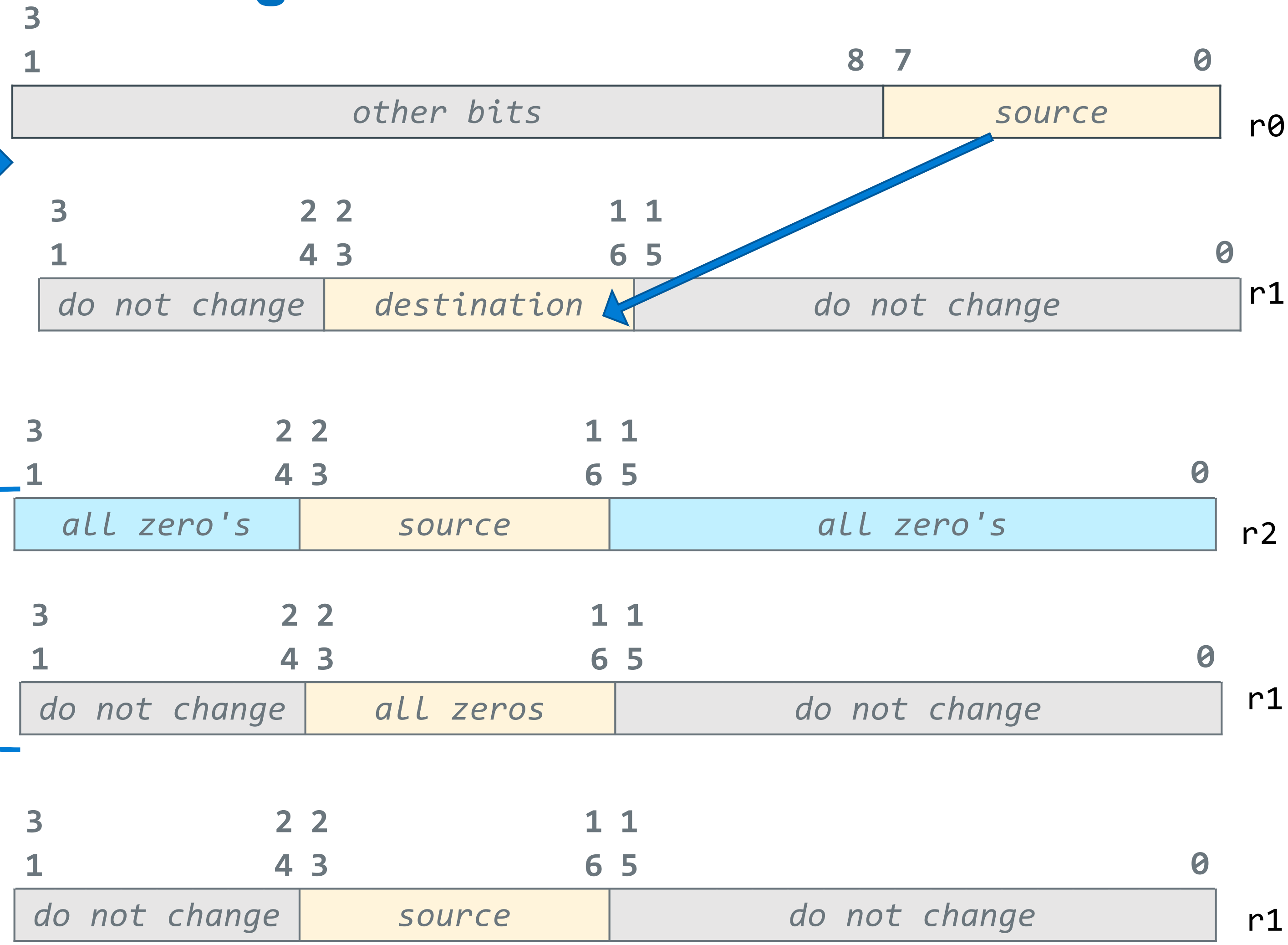
Task: Insert source into destination

a	b	a | b
0	0	0
0	1	1
1	0	1
1	1	1

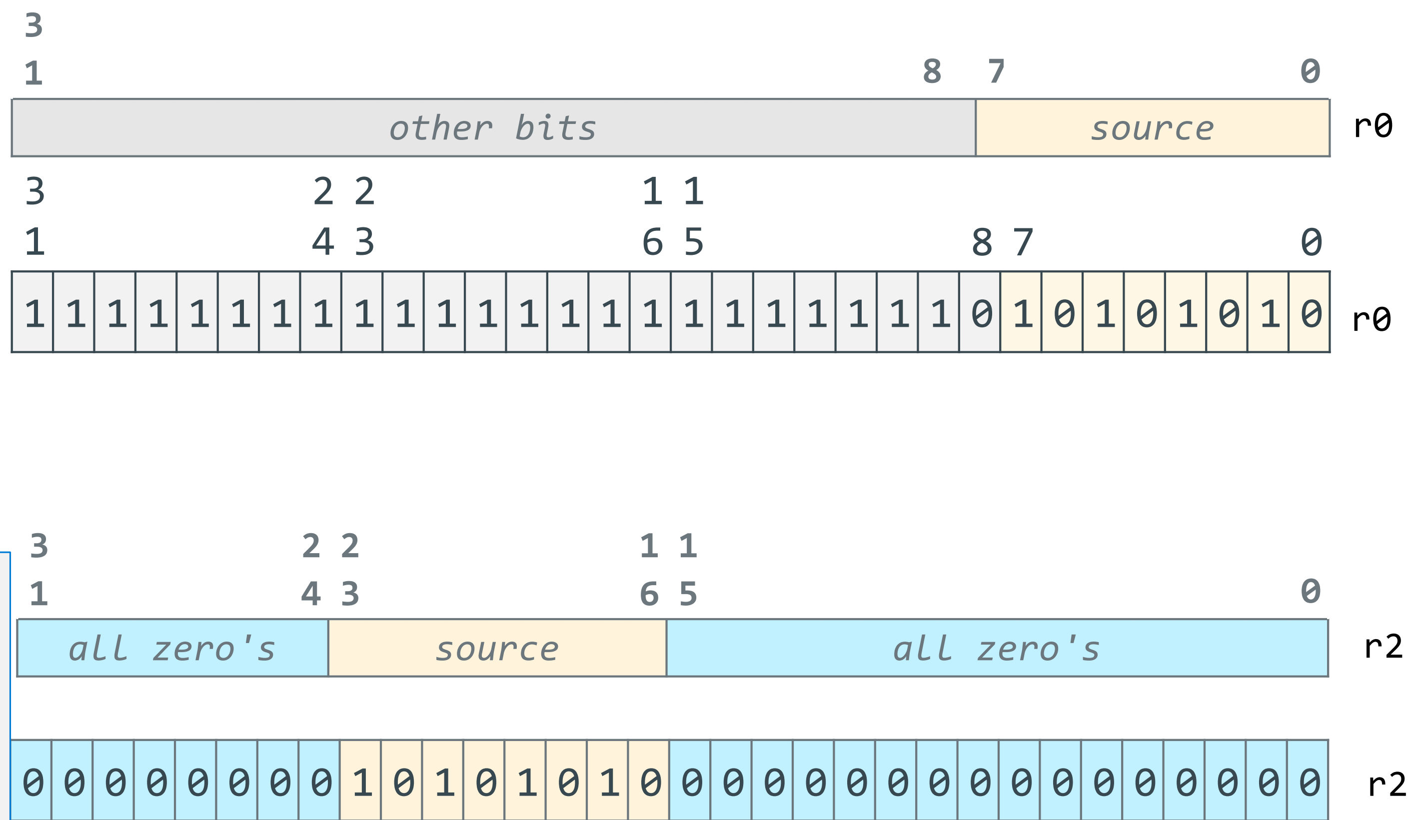
Approach
(1) isolate source field
(2) clear destination field
(3) Bitwise or together

```
orr    r1, r1, r2
r1 =   r1 | r2;
```

results in



Inserting Bitfields – Isolating the Source Field



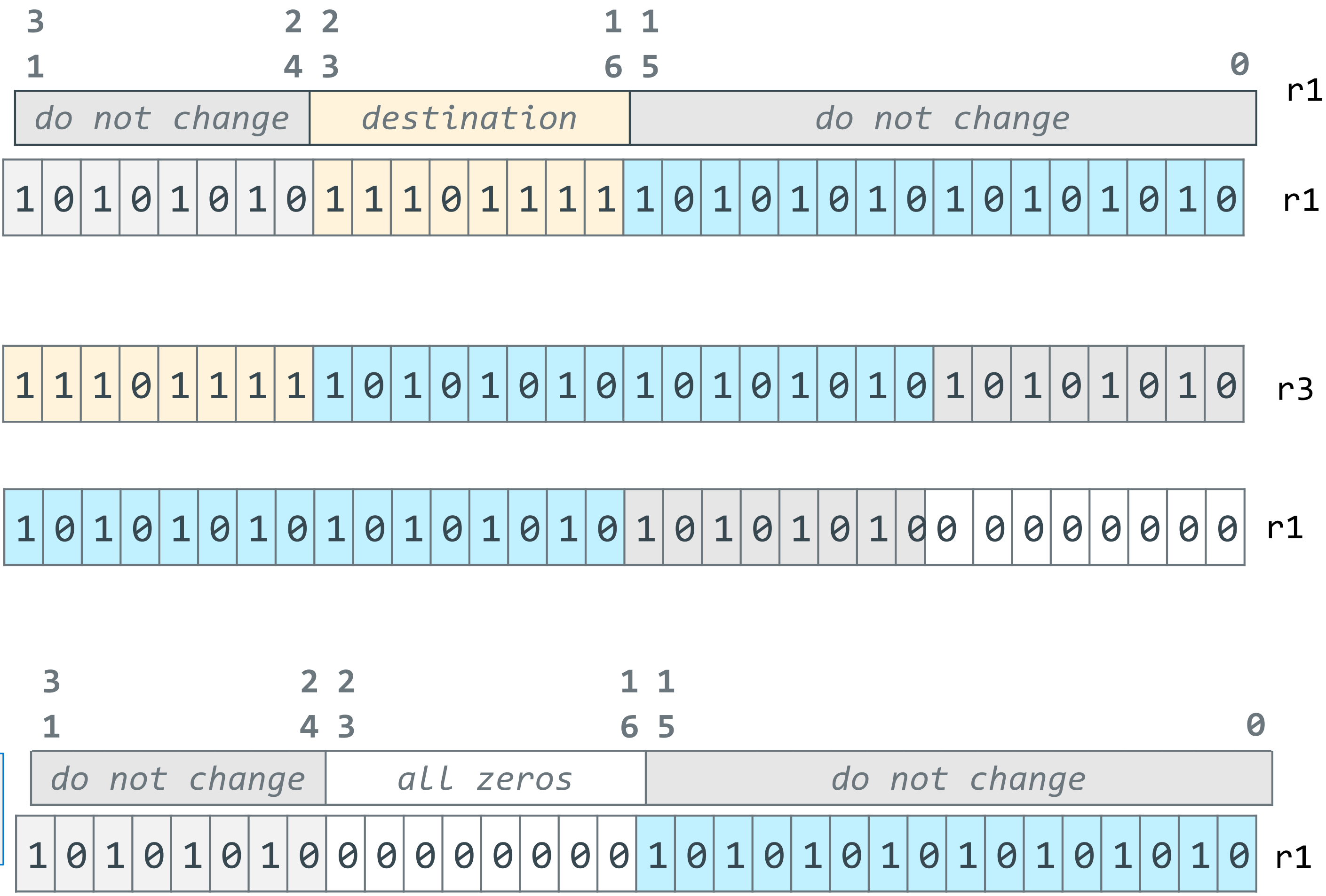

```
isolate source field

lsl    r2, r0, 24
lsr    r2, r2, 8

r2 = r0 << 24;
r2 = r2 >> 8;
```

Inserting Bitfields – Clearing the Destination Field

```
clear the
destination field
ror    r1, r1, 24
r1=(r1>>24)|(r1<<8);
```

```
lsl    r1, r1, 8
r1 = r1 << 8;
```

```
ror    r1, r1, 16
r1= (r1>>16)|(r1<<16);
```



x

Inserting Bitfields –
Combining Isolated Source and Cleared Destination

```
isolated source
```

31 … 24	23 … 16	15 … 0	
all zero's	*source*	*all zero's*	r2

0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	r2

```
field cleared in
destination
```

31 … 24	23 … 16	15 … 0	
do not change	*all zeros*	*do not change*	r1

1	0	1	0	1	0	1	0	0	0	0	0	0	0	0	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	r1

```
inserted field
orr    r1, r1, r0
r1 = r1 | r0;
```

1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	1	0	r1

X

Masking Summary

Select a field: Use `and` with a mask of one's surrounded by zero's to select the bits that have a 1 in the mask, all other bits will be set to zero

selects this field when used with and

0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0 0 0 0 0 0	selection mask

Clear a field: Use `and` with a mask of zero's surrounded by one's to select the bits that have a 1 in the mask, all other bits will be set to zero

clears this field when used with and

1 1 1 1 1 1 1 1	0 0 0 0 0 0 0 0 0	1 1 1 1 1 1 1 1 1 1 1 1 1 1 1	clear a field mask

Isolate a field: Use `lsr, lsl, rot` to get a field surrounded by zeros

0 0 0 0 0 0 0 0	*source*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0

lsl to get this edge into msb

lsr to get this edge into lsb

Insert a field: Use `orr` with fields surrounded by zeros

0 0 0 0 0 0 0 0	*source*	0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Keep these bits	0 0 0 0 0 0 0 0	Keep these bits

x

Extra Slides

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

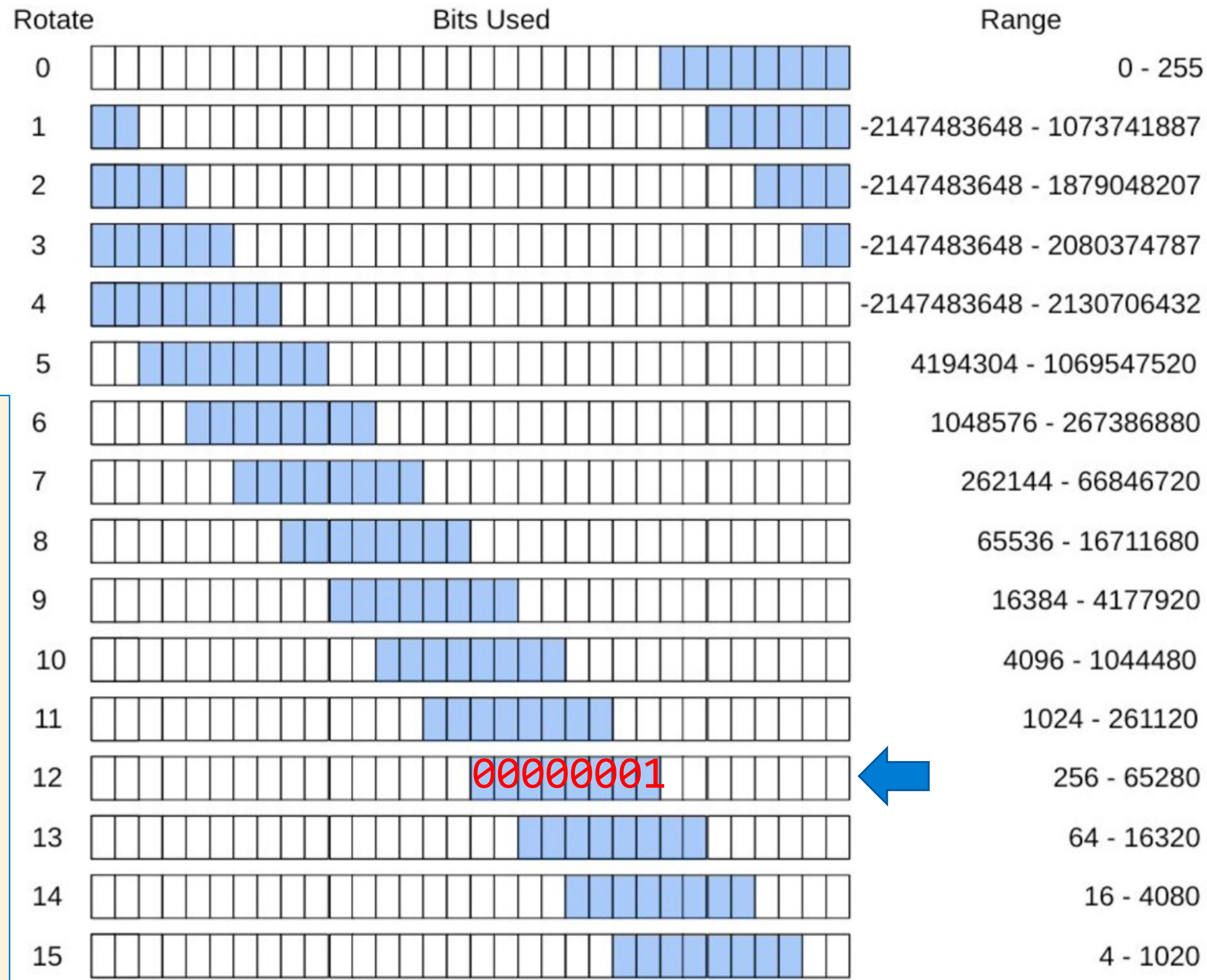
two bits rotated right

x

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: 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

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

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