3. Instruction Set Architecture – The LC2k and ARM architectures

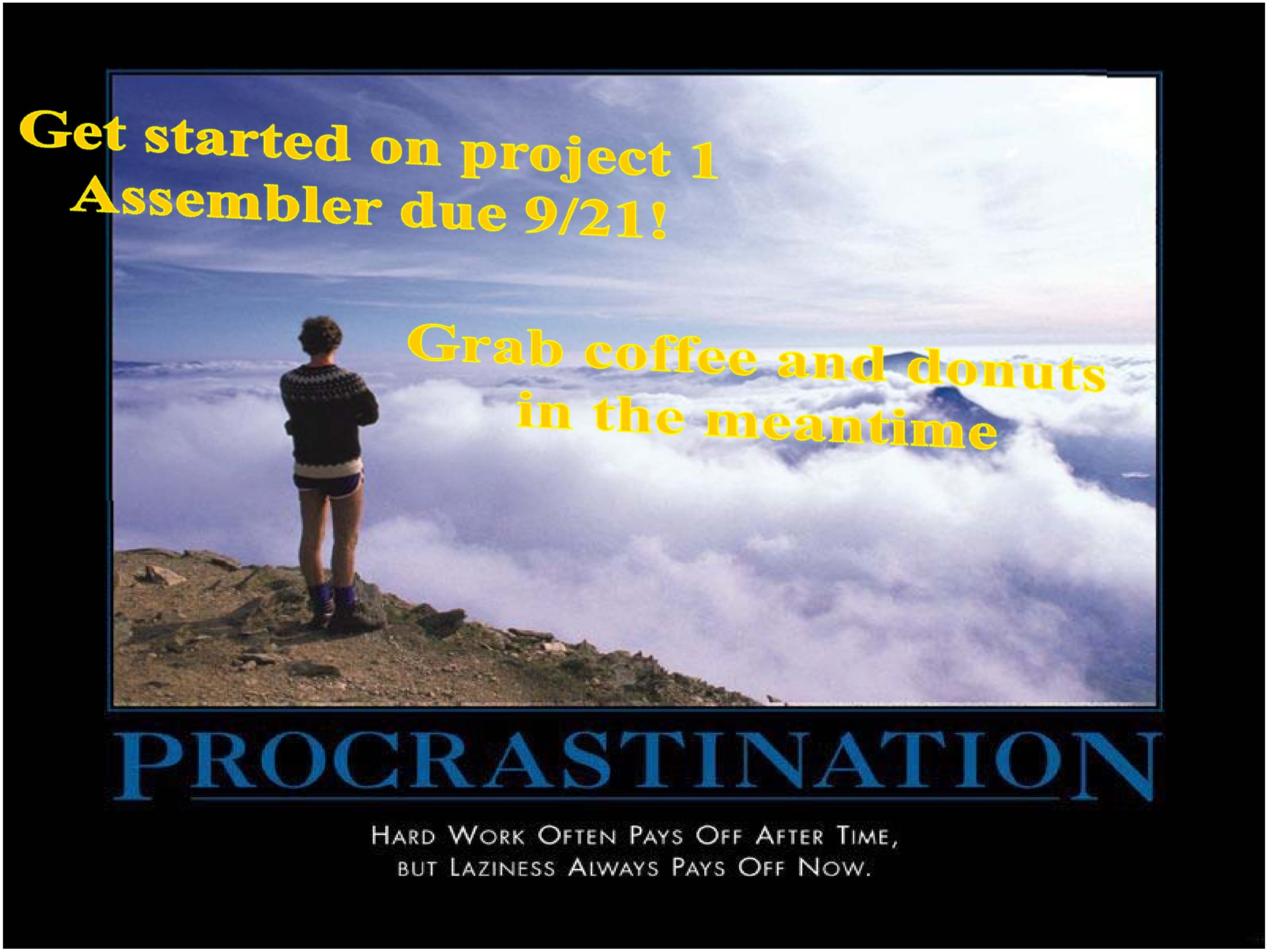
EECS 370 – Introduction to Computer Organization - Winter 2016

Profs. Valeria Bertacco & Reetu Das

EECS Department
University of Michigan in Ann Arbor, USA

Get started on project 1 Assembler due 9/21!

Grab coffee and donuts in the meantime


PROCRASTINATION
Hard Work Often Pays Off After Time,
but Laziness Always Pays Off Now.

Announcements

- Project 1
  - Project 1.a due 6:00pm, Thursday, 1/21
  - Project 1.m & 1.s due 6:00pm, Thursday, 2/4
- Homework 1 assigned
  - Due Thursday, 1/21
  - Submit electronically to www.gradescope.com, any electronic or scanned format is fine as long as the graders can find/read your answers, see homework sheet for submission details
- Discussions only on Friday 1/15, no classes on Monday 1/18
  - Please attend any Friday discussion, or review discussion materials on the class website and/or in office hours

Recap

- ISAs, instruction encoding
- Storage: registers, status register, memory, (accumulator)
- Addressing modes: direct, indirect, reg.indirect, base+displacement, pc-relative

Instruction Set Architecture (ISA) Design Lectures

- Lecture 2: Storage types and addressing modes
- Lecture 3 : LC-2K and ARM architecture
- Lecture 4 : Converting C to assembly – basic blocks
- Lecture 5 : Converting C to assembly – functions
- Lecture 6 : Translation software; libraries, VMs
- Lecture 7 : Memory layout

LC2K ISA

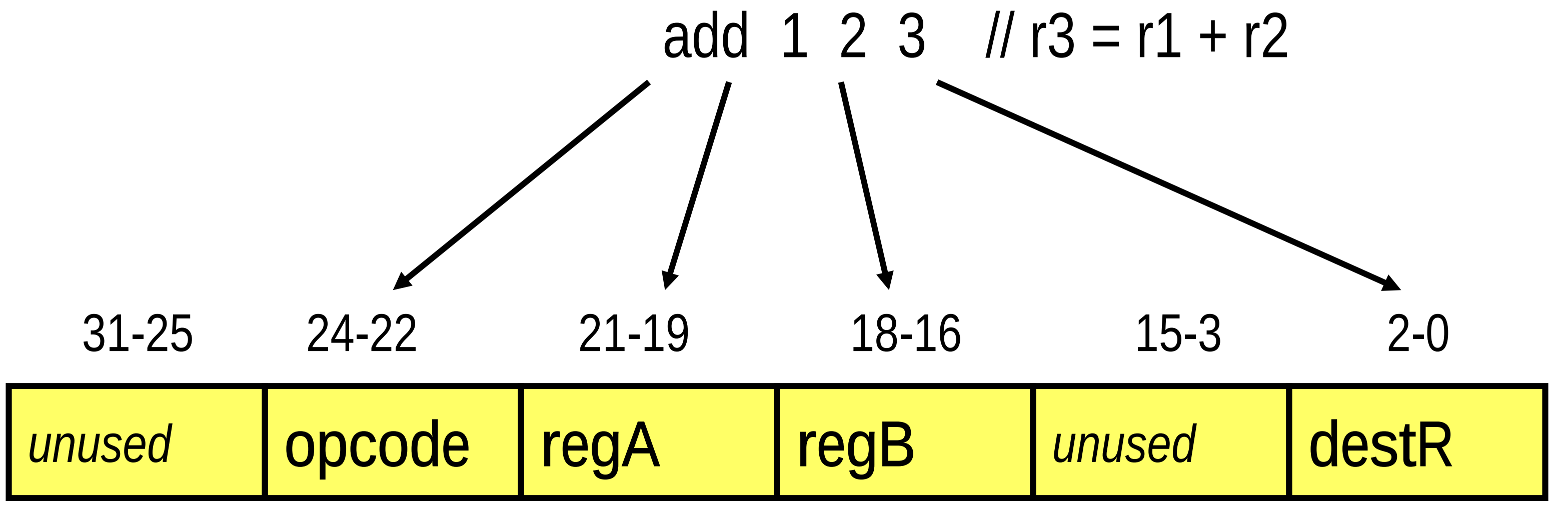
LC2K Processor

- 32-bit processor
  - Instructions are 32 bits
  - Integer registers are 32 bits
- 8 registers
- supports 65536 words of memory (addressable space)
- 8 instructions
  - add, nor, lw, sw, beq, jalr, halt, noop

LC2K ISA

Instruction Encoding

- Instruction set architecture defines the mapping of assembly instructions to machine code



LC2K ISA

Instruction Formats

- Positional organization of bits (Implies nothing about bit values!!!)

- R type instructions (add, nor)

31-25	24-22	21-19	18-16	15-3	2-0
unused	opcode	regA	regB	*unused*	destR

- I type instructions (lw, sw, beq)

31-25	24-22	21-19	18-16	15-0
unused	opcode	regA	regB	offset

LC2K ISA

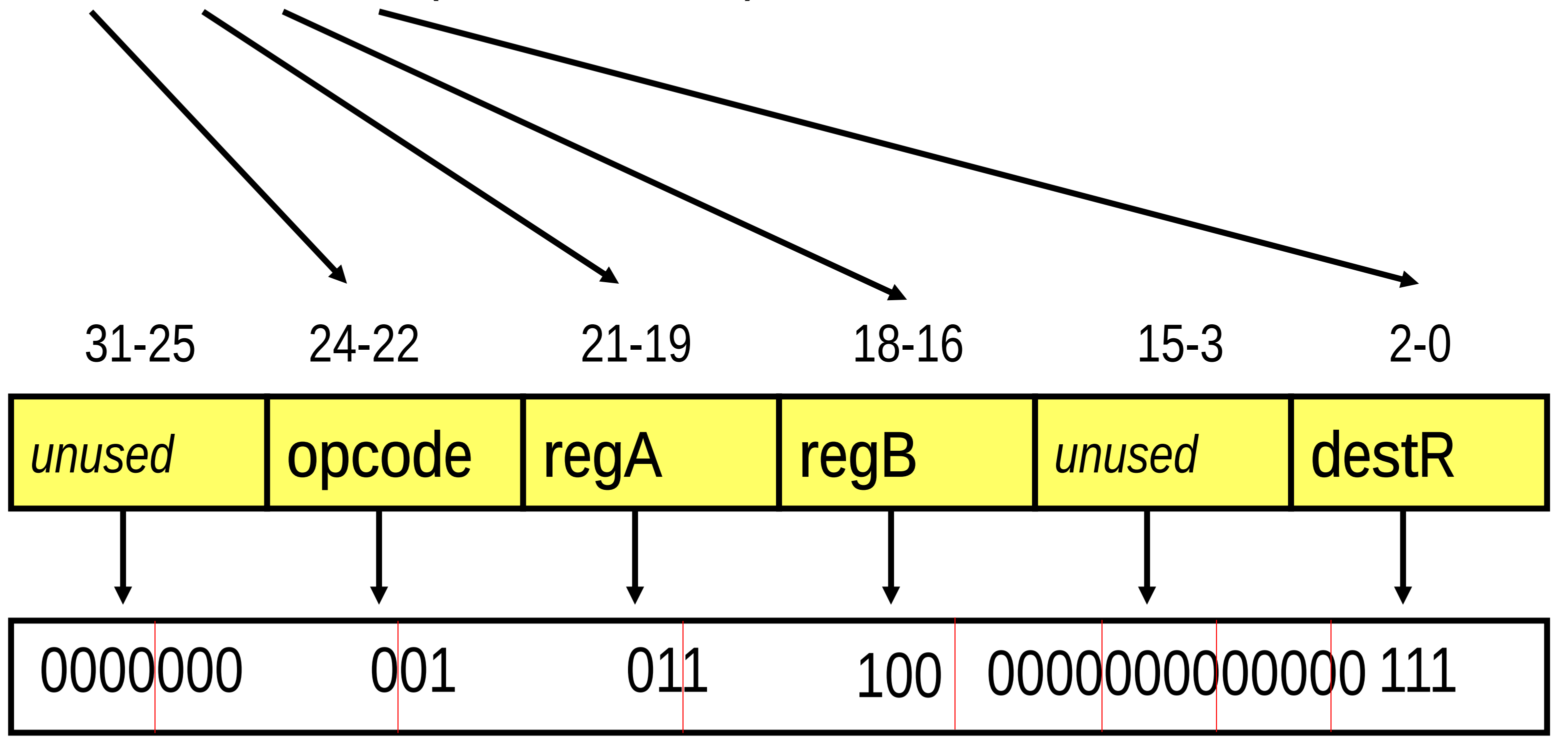
Bit Encodings

- Opcode encodings
  - add (000), nor (001), lw (010), sw (011), beq (100), jalr (101), halt (110), noop (111)
- Register values
  - Just encode the register number (r2 = 010)
- Immediate values
  - Just encode the values (Remember to give all the available bits a value!!)

LC2K ISA

Example Encoding - nor

- nor 3 4 7 (r7 = r3 nor r4)



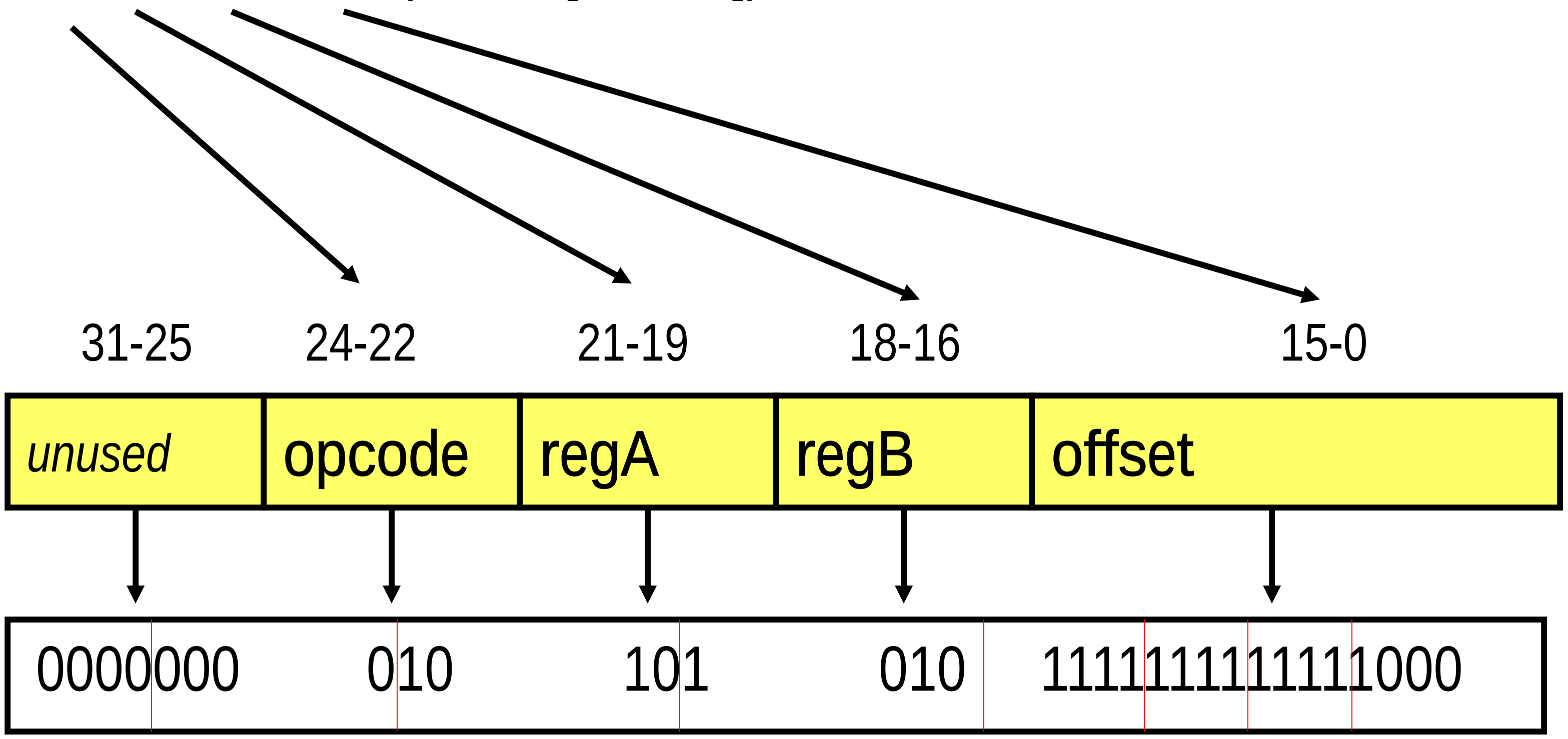
Convert to Hex → 0x005C0007

Convert to Dec → 6029319

LC2K ISA

Example Encoding - lw

- lw 5 2 -8 (r2 = M[r5 + -8])



Convert to Hex → 0x00AAFFF8
Convert to Dec → 11206648

LC2K ISA

Class Problem 1

- Compute the encoding in Hex for:
  - add 3 7 3 (r3 = r3 + r7) (add = 000)
  - sw 1 5 67 (M[r1+67] = r5) (sw = 011)

31-25	24-22	21-19	18-16	15-3	2-0
unused	opcode	regA	regB	*unused*	destR

31-25	24-22	21-19	18-16	15-0
unused	opcode	regA	regB	offset

ARM ISA

ARITH

ARM Instruction Set

- Three main types of instructions:
  1. Arithmetic
     - Add, subtract, multiply, multiply-accumulate
     - Logical: and, or, xor, shift, rotate, etc.
     - Compare : eq, ne, lt, gt, le, mi, pl, etc.
     - Data movement: mov, mvn
  2. Memory access
     - ldr, ldm (load multiple), str, stm (store multiple)
  3. Sequencing / control flow
     - b (branch), bl (branch and link)

ARM ISA ARITH

ARM Arithmetic Instructions

- Format: three operand fields
  - Dest. register usually the first one – check instruction format
  - e.g., add r3, r4, r7
  - Both SUB and RSB available
- C-code example: f = (g + h) – (i + j)

r3→g
r4→h
r5→i
r6→j
r7→f

```
add     t0, g, h
add     t1, i,  j
sub     f, t0, t1
```

$\longrightarrow$

```
add     r1, r3, r4
add     r2, r5, r6
sub     r7, r1, r2
```

ARM ISA
ARITH

ARM Arithmetic Instructions – special 2nd operand

Second source operand can be be

1. A **register** (see previous)

2. An **immediate** (8 bit) – optionally rotated by an immediate (4bits)
   - e.g., RSB r3, r4, #10
   - bit 25 of the instruction indicates whether reg or imm

   C-code example: f = 10 – g

```
rsb        r7, r3, #10
```

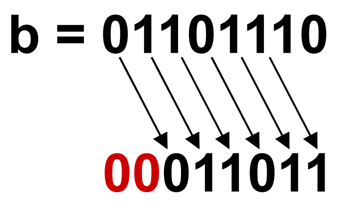
3. A **register to which a shift/rotate is applied first**
   - the shift amount can be specified in a register, or a 5-bit immediate

     e.g., MVN r5, r3 LSL #2   r5 = NOT (r3 << 2)

Opcode and mnemonic do not change

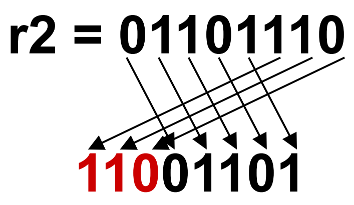
Mini-review: What is a *shift* ?

- C/C++
  - a = b >> 2;
  - c = d << 4;



- ARM
  - *dest* *src*
  - mov r3, r2, LSR #2
  - mov r5, r4, LSL #4

- Rotate also available in ARM
  - mov r3, r2, ROR #3



ARM ISA
ARITH

What About Immediates Larger Than 8 Bits??

- Use rotate right 0, 2, ..30 option on immediate operands
  - e.g., add r3, r4, #FF ROR 8 (equals #FF000000)
  - If this is not possible, load constant from memory into a register, then use a register operand
- Pseudo-instruction LDR rd, =K
- C-code example: f = f + 0x104

```
add     r7, r7, 0x41 ROR 30
```

(Note:ROR 30 = ROL 2)

$$\begin{tikzcd}\boxed{\text{0x041 (0b000001000001)}} \arrow[d] \\ \boxed{\text{0x104 (0b000100000100)}}\end{tikzcd}$$

ARM Arithmetic Instructions - recap

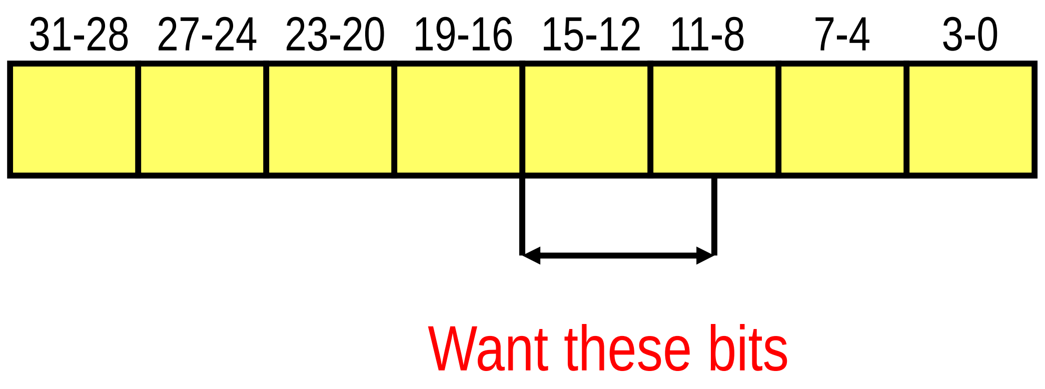
ARM ISA
ARITH

add	dest, src1, src2imm	// add
adc	dest, src1, src2imm	// add with carry
mul	dest, src1, src2	// multiply
sub	dest, src1, src2imm	// sub src2imm from src1
sbc	dest, src1, src2imm	// sub src2imm from src1 with carry
rsb	dest, src1, src2imm	// sub src1 from src2imm
rsc	dest, src1, src2imm	// sub src1 from src2imm with carry
and	dest, src1, src2imm	// logical AND of bits
orr	dest, src1, src2imm	// logical OR of bits
eor	dest, src1, src2imm	// exclusive OR of bits
mov	dest, src1imm	// mov src1imm into dest (can be shifted)
mvn	dest, src1imm	// mov NOT(src1imm) to dest (shift ok)
cmp	src1, src2imm	// compare src1 to src2imm & set PSR

ARM ISA
ARITH

Class Problem 2

- Show the C and ARM assembly for extracting the value in bits 15:10 from a 32-bit integer variable



Assume the variable is in r1

Remember each hex digit is 4 bits

ARM Memory Instructions

ARM ISA

MEM

- Supports base + displacement and base + register modes
  - Base is a register
  - Displacement is a 12-bit immediate, positive or negative
  - Register displacement can be shifted/rotated (5-bits), immediate cannot
  - Pre-indexed and post-indexed offset modes allow to modify the base register before or after the memory access
- Examples:

```
ldr   r3,  [r4, #1000]   // load word: r3= M[r4+1000]
ldr   r3,  [r4, -r1, LSL #2]   // load word: r3= M[r4-(r1<<2)]
str   r3,  [r4, # -34]!   // load word pre-ndx: M[r4-34]= r3; r4-=34
ldr   r3,  [r4], # 20   // load word post-ndx: r3= M[r4]; r4+=20
```

ARM ISA
MEM

Load Instruction Sizes

How much data is retrieved from memory at the given address?

- ldr r3, [r4, #1000]
  - retrieve a word (32 bits) from address (r4+1000)
- ldrh r3, [r4, #1000]
  - retrieve a halfword (16 bits) from address (r4+1000)
- ldrb r3, [r4, #1000]
  - retrieve a byte (8 bits) from address (r4+1000)

Sign/Zero Extension

- Registers in ARM are 32 bits!
- So what happens when you load 8 or 16 bits?
  - Sign extend if the load is signed

$$\boxed{\texttt{0x1F}} \rightarrow \boxed{\texttt{0x0000001F}} \qquad \boxed{\texttt{0xFE}} \rightarrow \boxed{\texttt{0xFFFFFFFE}}$$

  - Zero extend if the load is unsigned (default)

$$\boxed{\texttt{0x1F}} \rightarrow \boxed{\texttt{0x0000001F}} \qquad \boxed{\texttt{0xFE}} \rightarrow \boxed{\texttt{0x000000FE}}$$

ARM Memory Instructions - recap

- Load instructions

ldrsb	\\ load byte signed (load 8 bits, sign extend)
ldrb	\\ load byte unsigned (load 8 bits, zero extend)
ldrsh	\\ load halfword (load 16 bits, sign extend)
ldrh	\\ load halfword unsigned (load 16 bits, zero extend)
ldr	\\ load word (load 32 bits, no extension)

- Store instructions (No sign/zero extension for stores)

strb r3, [r4, #1000]	\\ store 8 LSBs of r3 to M[r4+1000]
strh r3, [r4, #1000]	\\ store 16 LSBs of r3 to M[r4+1000]
str r3, [r4, #1000]	\\ store all 32 bits of r3 to M[r4+1000]

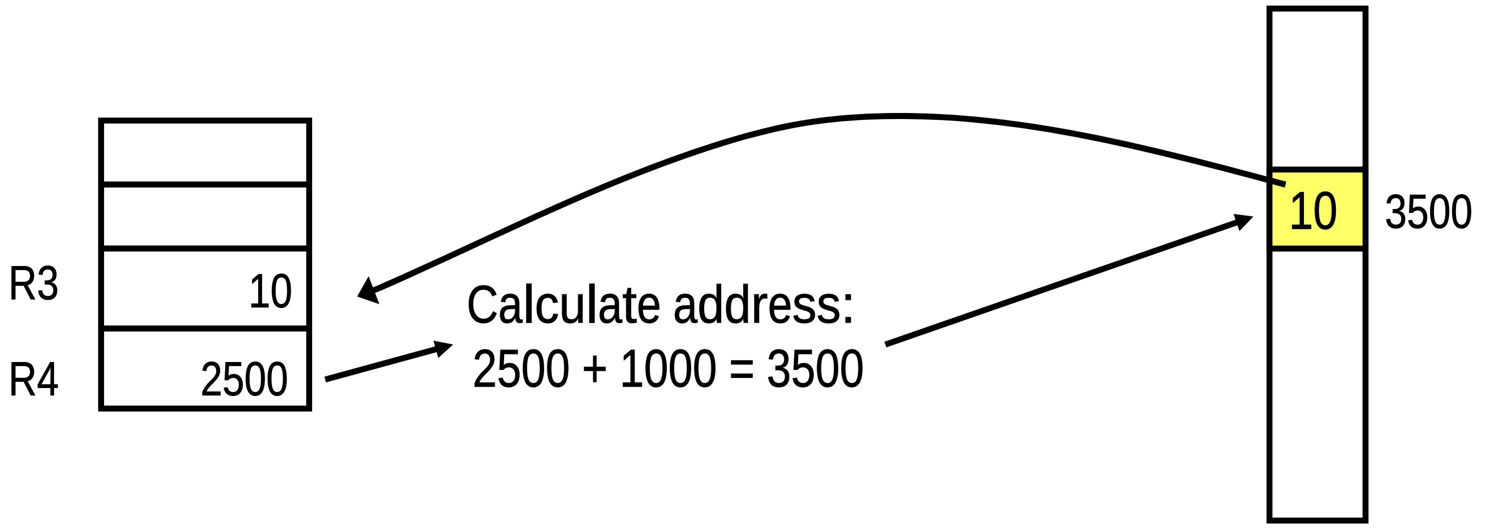
- Addressing: base+displ; displ = {reg, imm, reg w. shift/rot}
- Pre- and post-indexing

ARM ISA MEM

Load Instruction in Action

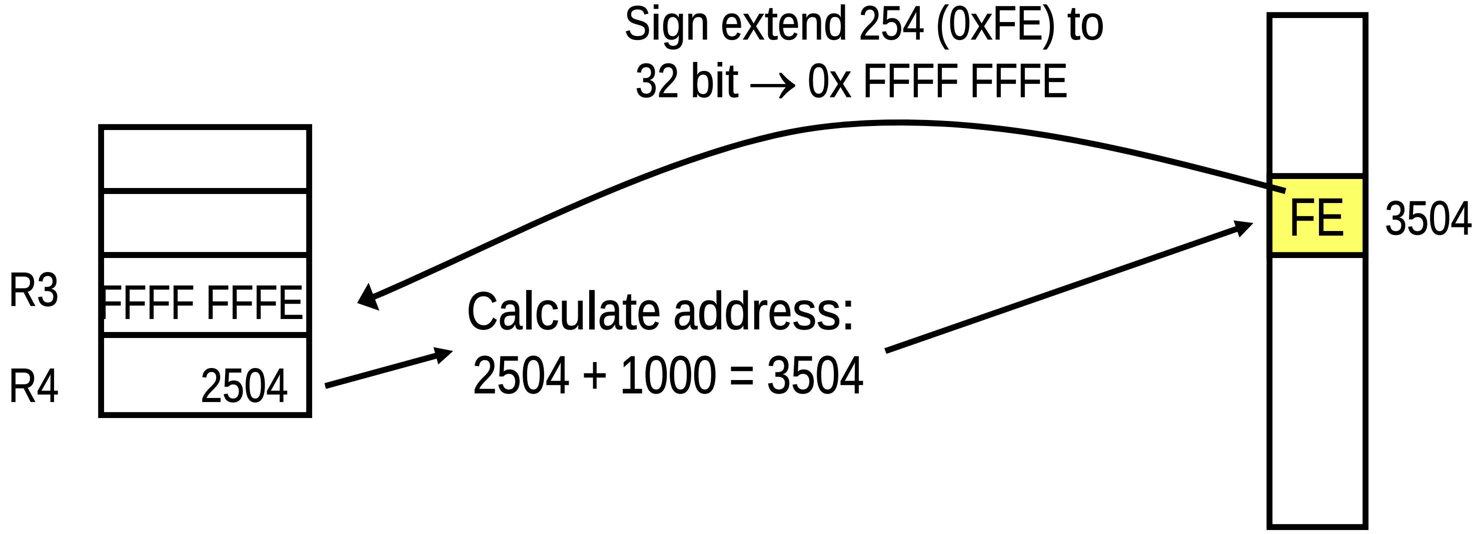
- ldrsb r3, [r4, #1000] // load signed byte

  Retrieves 8-bit value from memory location (r4+1000) and puts the result into r3 (sign extended)



Load Instruction in Action – other example

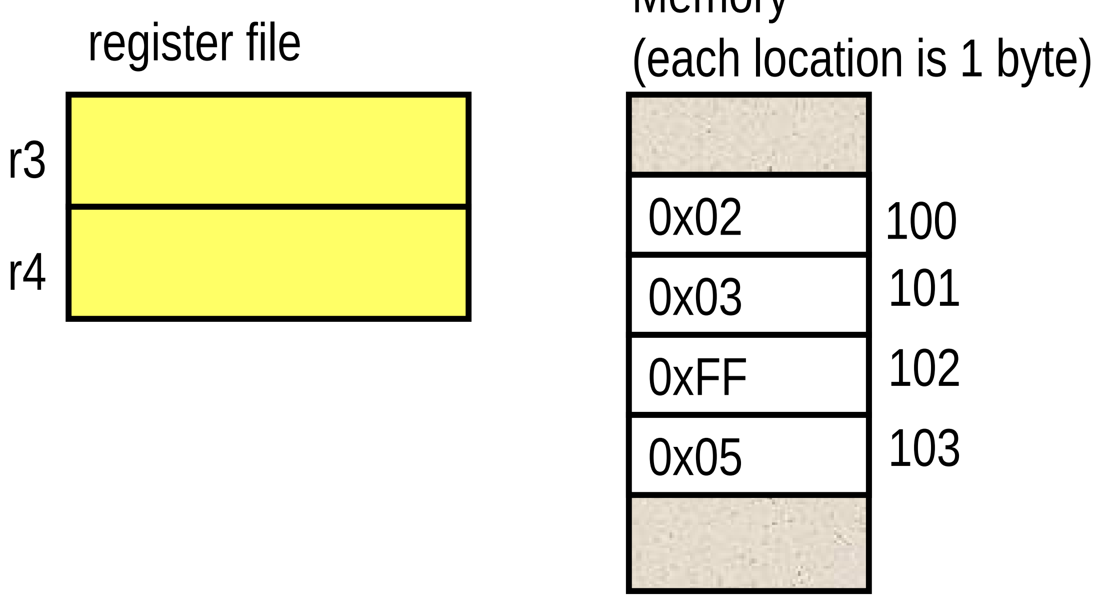
- ldrsb r3, [r4, #1000] // load signed byte
  Retrieves 8-bit value from memory location (r4+1000) and puts the result into r3 (sign extended)



Example Code Sequence

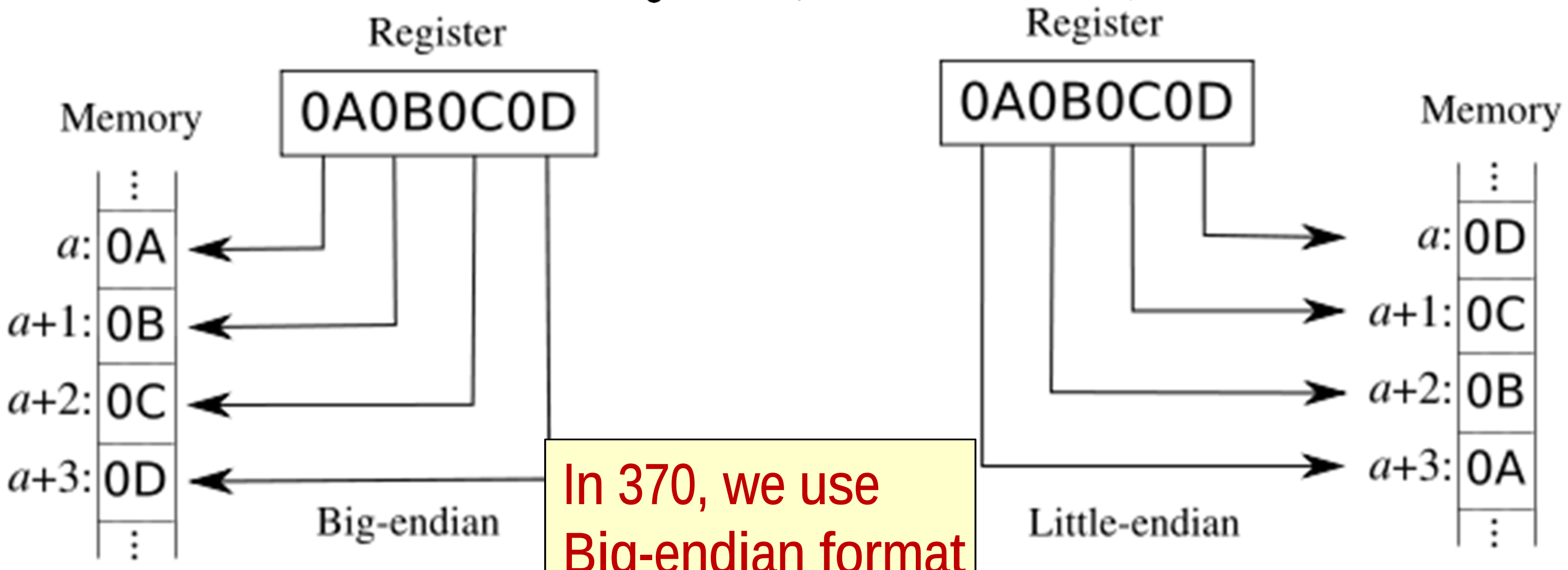
What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr      r4, [r0, #100]
ldrsb    r3, [r0, #102]
str      r3, [r0, #100]
strb     r4, [r0, #102]
```



Important note on addressing: Big-endian vs. Little-endian

- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big – The opposite, most significant byte first
  - Internet and SPARC are big endian, x86 is little endian, ARM is bi-endian



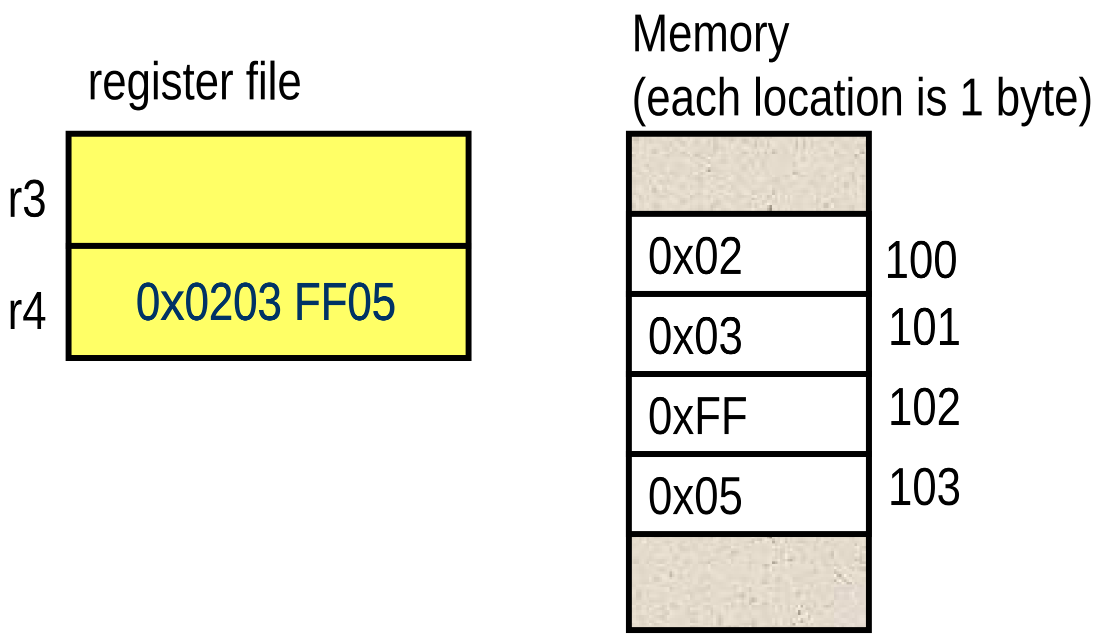
In 370, we use Big-endian format

ARM ISA
MEM

Example Code Sequence – insn 1

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```



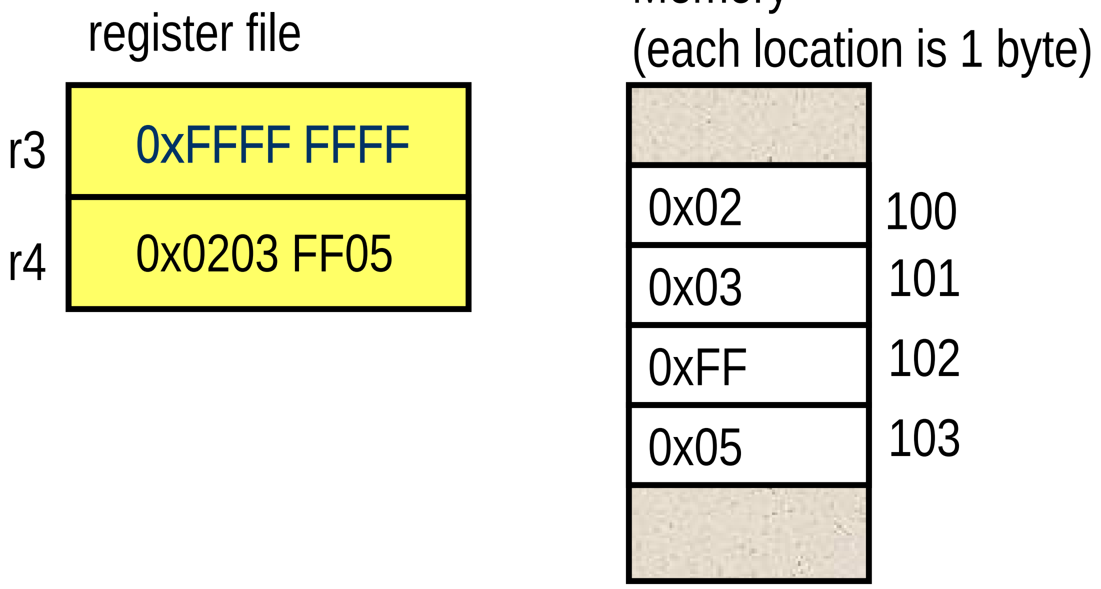
ARM ISA

MEM

Example Code Sequence – insn 2

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr      r4, [r0, #100]
ldrsb    r3, [r0, #102]
str      r3, [r0, #100]
strb     r4, [r0, #102]
```

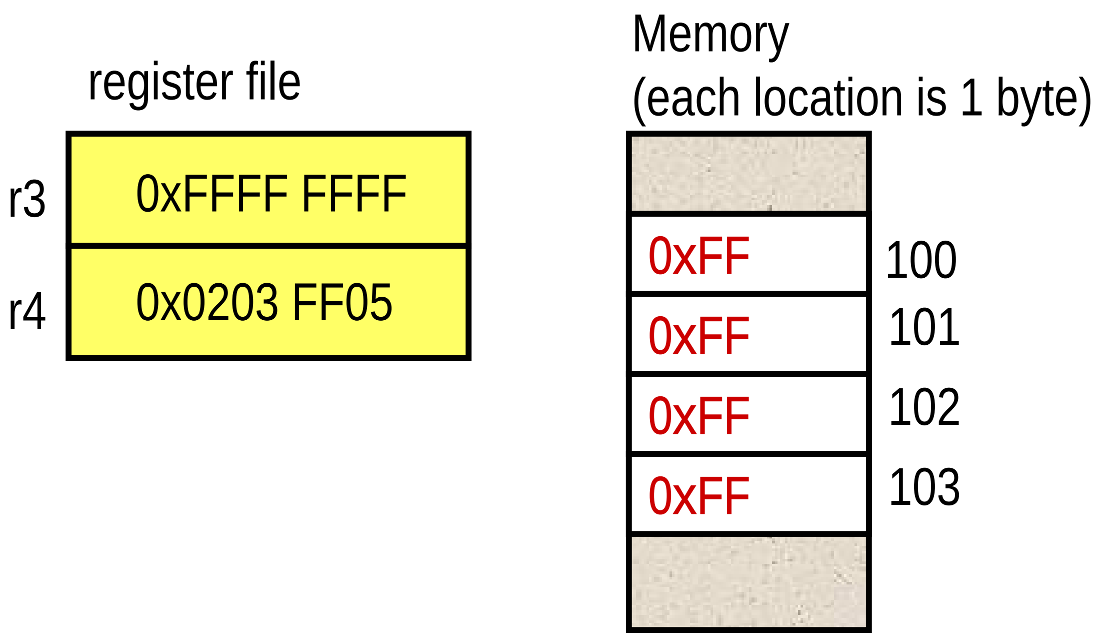


Example Code Sequence – insn 3

ARM ISA MEM

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

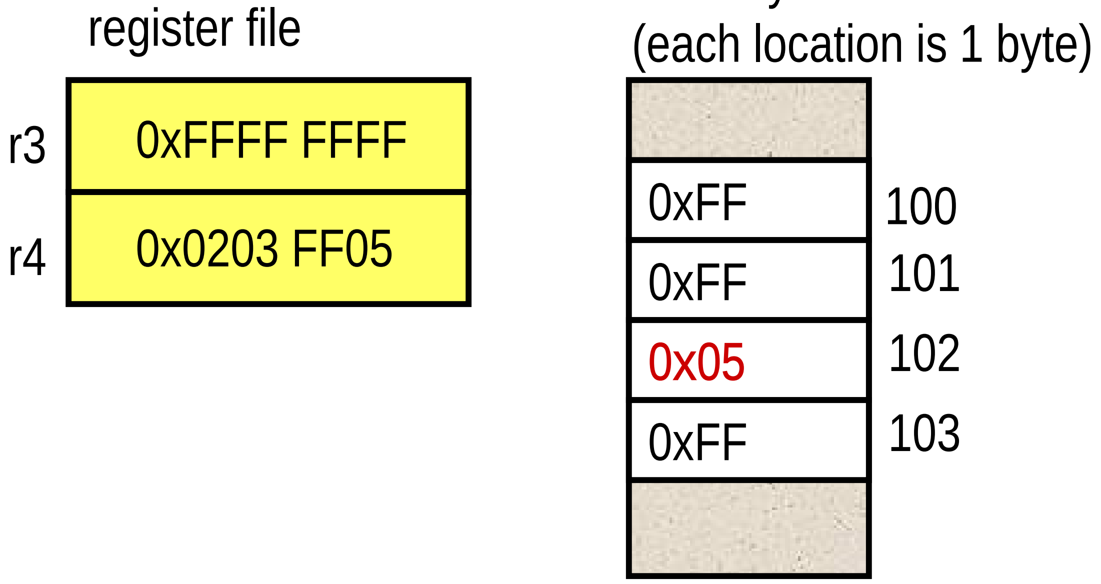
```
ldr     r4, [r0, #100]
ldrsb   r3, [r0, #102]
str     r3, [r0, #100]
strb    r4, [r0, #102]
```



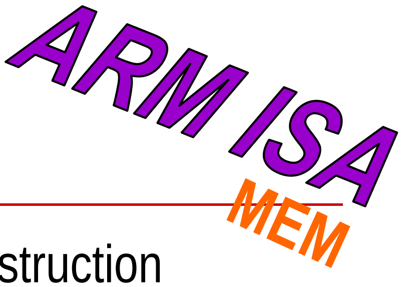
Example Code Sequence – insn 4

What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldr      r4, [r0, #100]
ldrsb    r3, [r0, #102]
str      r3, [r0, #100]
strb     r4, [r0, #102]
```

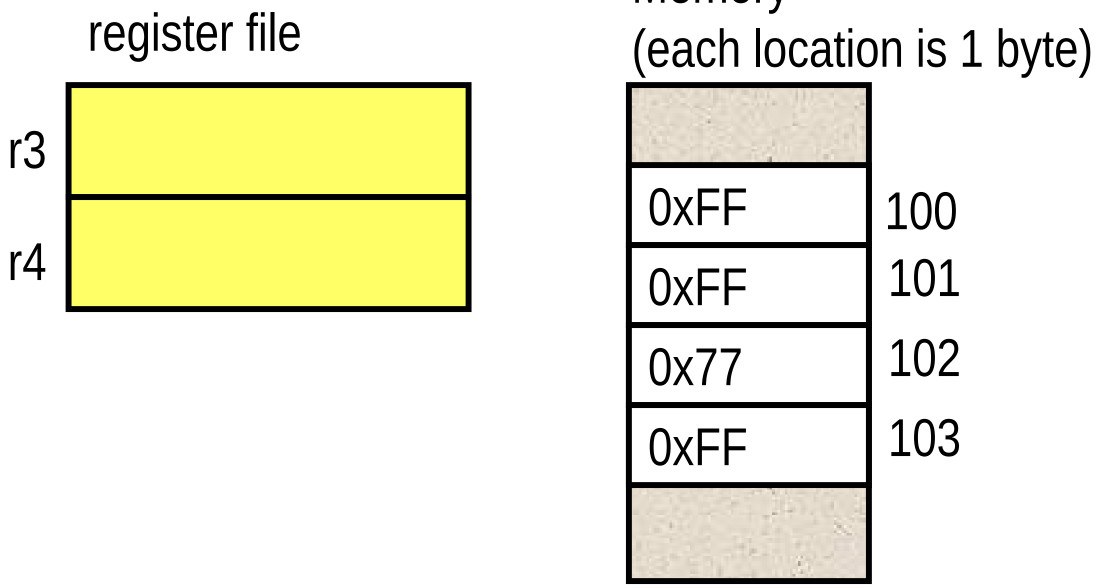


Class Problem 3



What is the final state of memory once you execute the following instruction sequence? (assume r0 stores the value 0)

```
ldrh    r3, [r0, #100]
ldrb    r4, [r0, #102]
str     r3, [r0, #100]
strh    r4, [r0, #102]
```



ARM Sequencing Instructions

ARM ISA
BRANCH

- Sequencing instructions change the flow of instructions that are executed
  - This is achieved by modifying the program counter (r15)
- Conditional branches
  - If (condition_test) goto target_address
    - *condition_test* examines flags from the processor status word (PSR)
    - *target_address* is a 24-bit word displacement on current PC+8

- cmp r1, r2
  beq label
- if (r1 == r2) then PC = label else PC = PC + 4

ARM ISA
BRANCH

ARM Condition Codes Determine Branch Direction

- Most arithmetic/logic instructions can set condition codes in PSR
  - add, sub, cmp, and, eor, etc…
  - You need to add “S” to the instruction: adds, subs, ands,…
- Four primary condition codes evaluated:
  - N – set if the result is negative (i.e., bit 31 is non-zero)
  - Z – set if the result is zero (i.e., all 32 bits are zero)
  - C – set if last addition/subtraction had a carry/borrow out of bit 31
  - V – set if the last addition/subtraction produced an overflow (e.g., two negative numbers added together produced a positive result)
- Branch conditions:

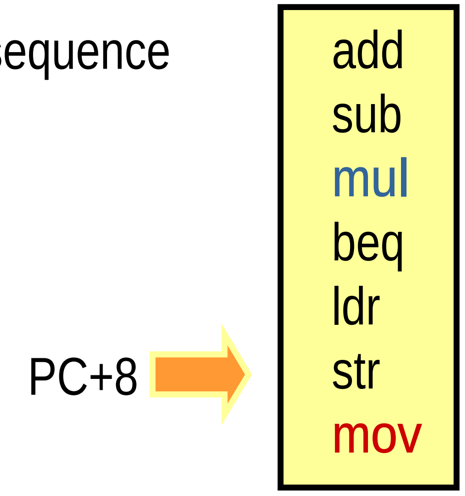
eq – (Z == 1)	gt – (Z == 0 && N == V)	mi - ?
ne – (Z == 0)	le – (Z == 1 || N != V)	pl – ?
ge – (N == V)	lt – (N != V)	al – 1 (can use shorthand “b label”)

ARM ISA
BRANCH

Setting the Branch Displacement Field

- Target address is a 24-bit signed aligned displacement on current PC+8
  - Target = PC + 8 + 4 * 24_bit_signed_displacement
    - beq 1 // branch 3 instructions ahead if flag Z == 1
    - beq -3 // branch 1 instruction back if flag Z == 1
    - beq -2 // Infinite loop if flag Z == 1

Example code sequence



ARM ISA
BRANCH

Other Branching Instructions

- cmp r2, r3 // branch to 4*offset+PC+8 if r2 $\neq$ r3
  bne offset
- cmp r2, r3 // branch to 4*offset+PC+8 if r2 < r3
  blt offset
- cmp r2, r3 // branch to 4*offset+PC+8 if r2 $\geq$ r3
  bge offset

- b offset // jump to 4*offset+PC+8
  // unconditional jump, is taken regardless of PSR flags

- mov r15, r3 // jump to address in r3 -- when is this useful?

- bl offset // put PC+4 into register r14 (LR) and jump to 4*offset+PC+8

Branch - Example

ARM ISA
BRANCH

Convert the following C code into ARM assembly (assume x is in r1, y in r2):

```
     int x, y;
     if (x == y)
          x++;
(L1) else
          y++;
(L2) ...
```

Using Labels

```
     cmp  r1, r2
     bne  L1
     add r1, r1, #1
     b L2
L1:  add r2, r2, #1
L2:  ...
```

Without Labels

```
cmp r1, r2
bne 1
add r1, r1, #1
b 0
add r2, r2, #1
```

Assemblers should deal with labels and assign displacements – why?