3. Instruction Set Architecture – The LC2k and ARM architectures

EECS 370 – Introduction to Computer Organization – Winter 2015

Robert Dick, Andrew Lukefahr, and Satish Narayanasamy

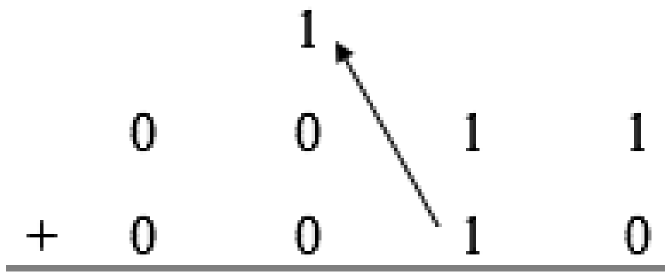
EECS Department
University of Michigan in Ann Arbor, USA

© Dick-Lukefahr-Narayanasamy, 2015
The material in this presentation cannot be copied in any form without our written permission

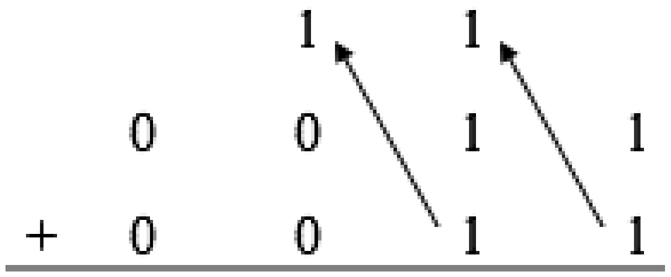
Representing negative numbers

- We already know how to represent non-negative numbers in binary
  - 6 (base 10)
    = 4 + 2 = 1*4+ 1*2 + 0*1 = 1*2^2 + 1*2^1 + 0*2^0 = 110 (base 2)
- And we can do arithmetic with binary numbers:

3 + 2 = 5



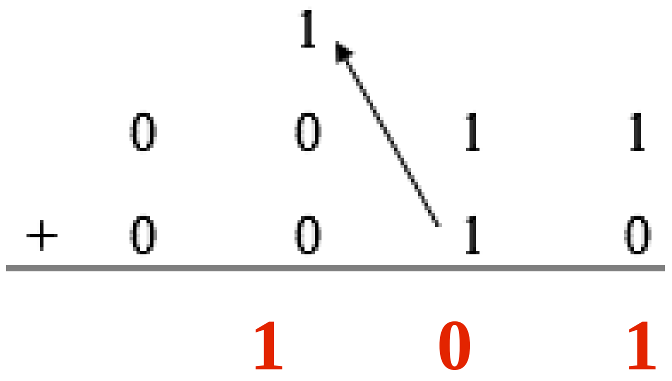
3 + 3 = 6



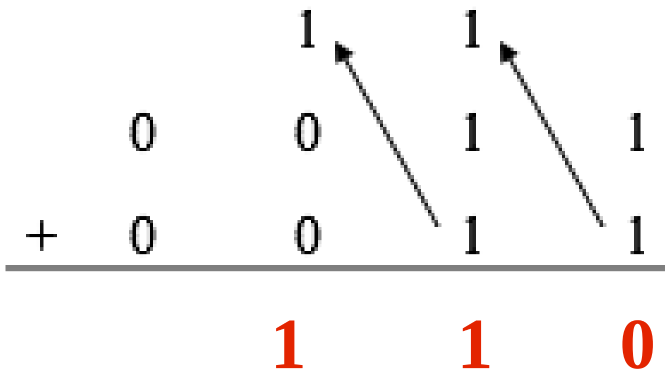
Representing negative numbers

- We already know how to represent non-negative numbers in binary
  - 6 (base 10)
    = 4 + 2 = 1*4+ 1*2 + 0*1 = 1*2^2 + 1*2^1 + 0*2^0 = 110 (base 2)
- And we can do arithmetic with binary numbers:

3 + 2 = 5



3 + 3 = 6



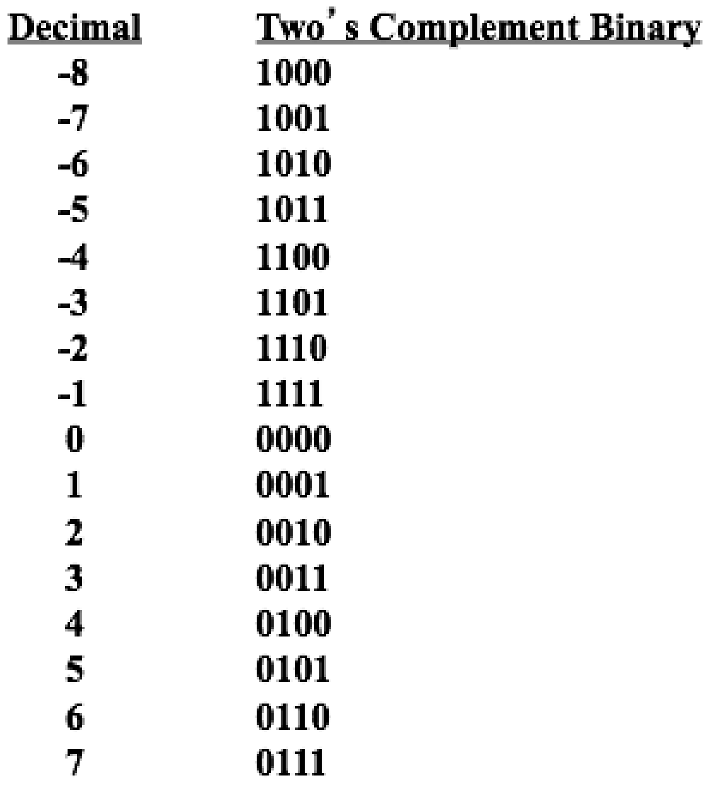
Negative Numbers?

- We would like a number system that provides
  - obvious representation of 0,1,2...
  - uses adder for addition
  - single value of 0
  - equal coverage of positive and negative numbers
  - easy detection of sign
  - easy negation

Two’s Complement Representation

- 2’s complement representation of negative numbers
- Take the bitwise inverse and add 1
- Biggest 4-bit Binary Number: 7
- Smallest 4-bit Binary Number: -8

- Exercise: with n bits, what are the smallest and largest 2’s complement numbers?



Decimal	Two’s Complement Binary
-8	1000
-7	1001
-6	1010
-5	1011
-4	1100
-3	1101
-2	1110
-1	1111
0	0000
1	0001
2	0010
3	0011
4	0100
5	0101
6	0110
7	0111

Two’s Complement Representation

- Negate:
  - Take the bitwise inverse and add 1
- Sign extension
  - +3 = 0011 = 00000011 = 0000000000000011
  - -3 = 1101 = 11111101 = 1111111111111101

- Exercise: negate zero
- Exercise: represent -23 with the minimum number of bits
- Exercise: represent -23 as an 8-bit binary number

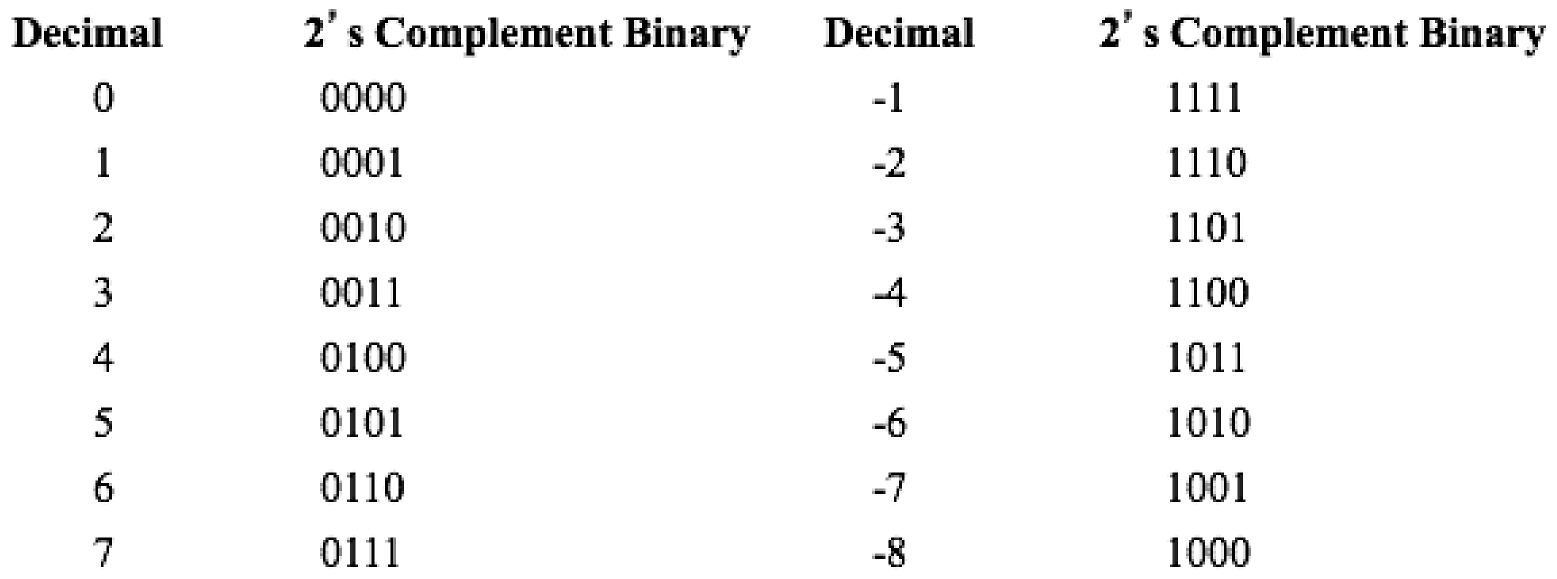
Two’s Complement Arithmetic (Subtraction)

Decimal	2’s Complement Binary	Decimal	2’s Complement Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

Examples: 7 - 6 = 7 + (- 6) = 1 3 - 5 = 3 + (- 5) = -2

$$\begin{array}{rcccc} & 0 & 1 & 1 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline \end{array} \qquad \begin{array}{rcccc} & 0 & 0 & 1 & 1 \\ + & 1 & 0 & 1 & 1 \\ \hline \end{array}$$

Two’s Complement Arithmetic (Subtraction)



Decimal	2’s Complement Binary	Decimal	2’s Complement Binary
0	0000	-1	1111
1	0001	-2	1110
2	0010	-3	1101
3	0011	-4	1100
4	0100	-5	1011
5	0101	-6	1010
6	0110	-7	1001
7	0111	-8	1000

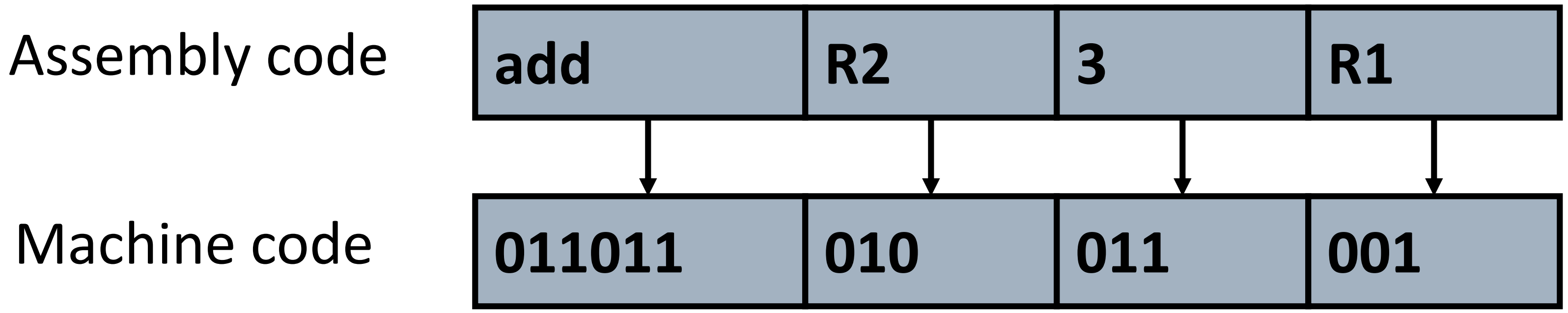
Examples: 7 - 6 = 7 + (- 6) = 1   3 - 5 = 3 + (- 5) = -2

$$\begin{array}{ccccc} & 1 & 1 & & \\ & 0 & 1 & 1 & 1 \\ + & 1 & 0 & 1 & 0 \\ \hline & 0 & 0 & 0 & 1 \end{array} \qquad \begin{array}{ccccc} & & 1 & 1 & \\ & 0 & 0 & 1 & 1 \\ + & 1 & 0 & 1 & 1 \\ \hline & 1 & 1 & 1 & 0 \end{array}$$

Recap (Assembly/Machine Code)

- Computers store program instructions the same way they store data.

- Each instruction is encoded as a number
  - Opcode field: what instruction to perform.
  - Operand fields: what data to perform it on.



Recap (Storage)

- Registers
  - Small array of storage locations in processor
  - Fast access
  - Direct addressing only
  - Special register: the PC register

- Memory
  - Large array of storage locations
  - Slow access
  - Many addressing modes (direct, indirect, reg.indirect, base+displacement)

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

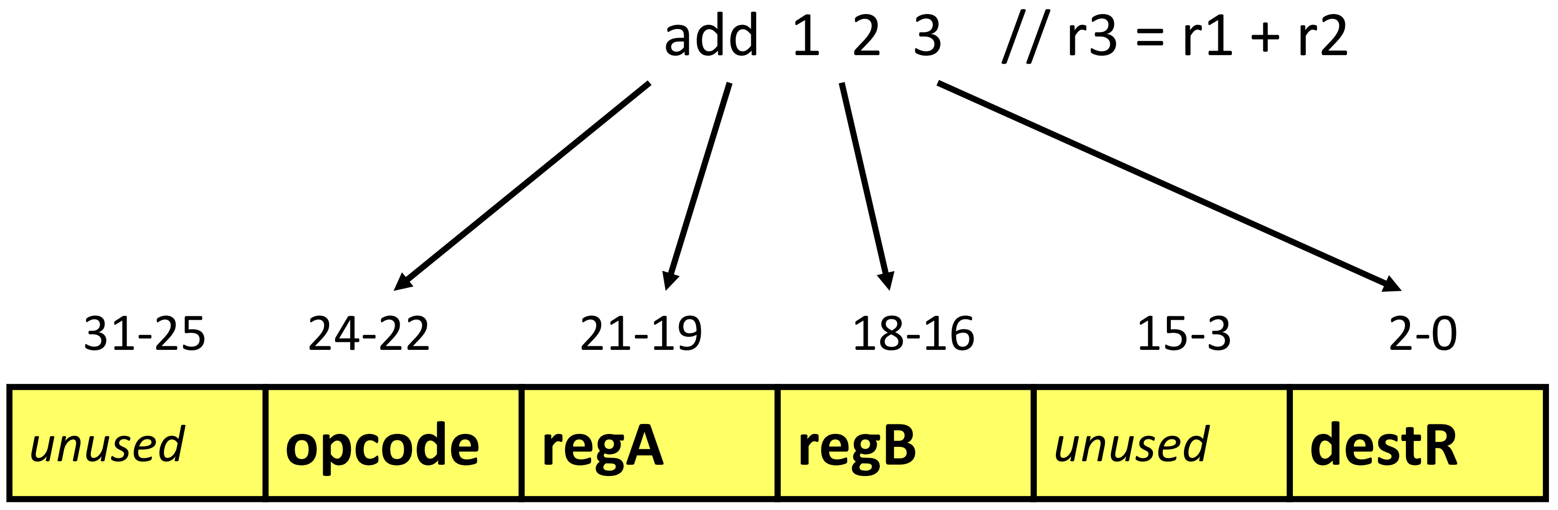
LC2K Processor

- 32-bit processor
  - Instructions are 32 bits
  - Integer registers are 32 bits
- 8 registers
- supports 65536 words of memory (addressable space)
- 8 instructions
  - add, nand, lw, sw, beq, jalr, halt, noop

Instruction Encoding

LC2K ISA

- 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, nand)

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

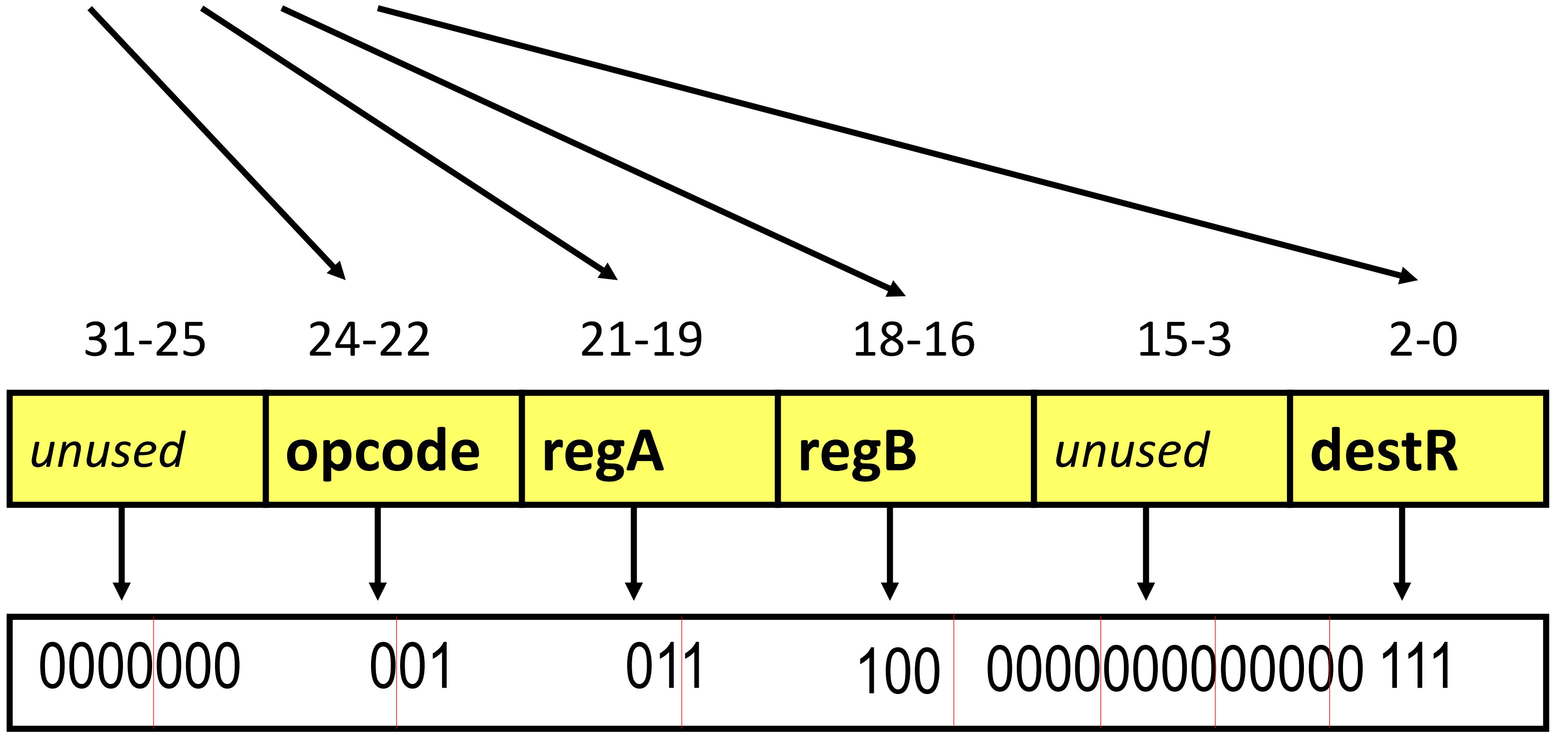
Bit Encodings

- Opcode encodings
  - add (000), nand (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 - nand

- nand 3 4 7 (r7 = r3 nand r4)



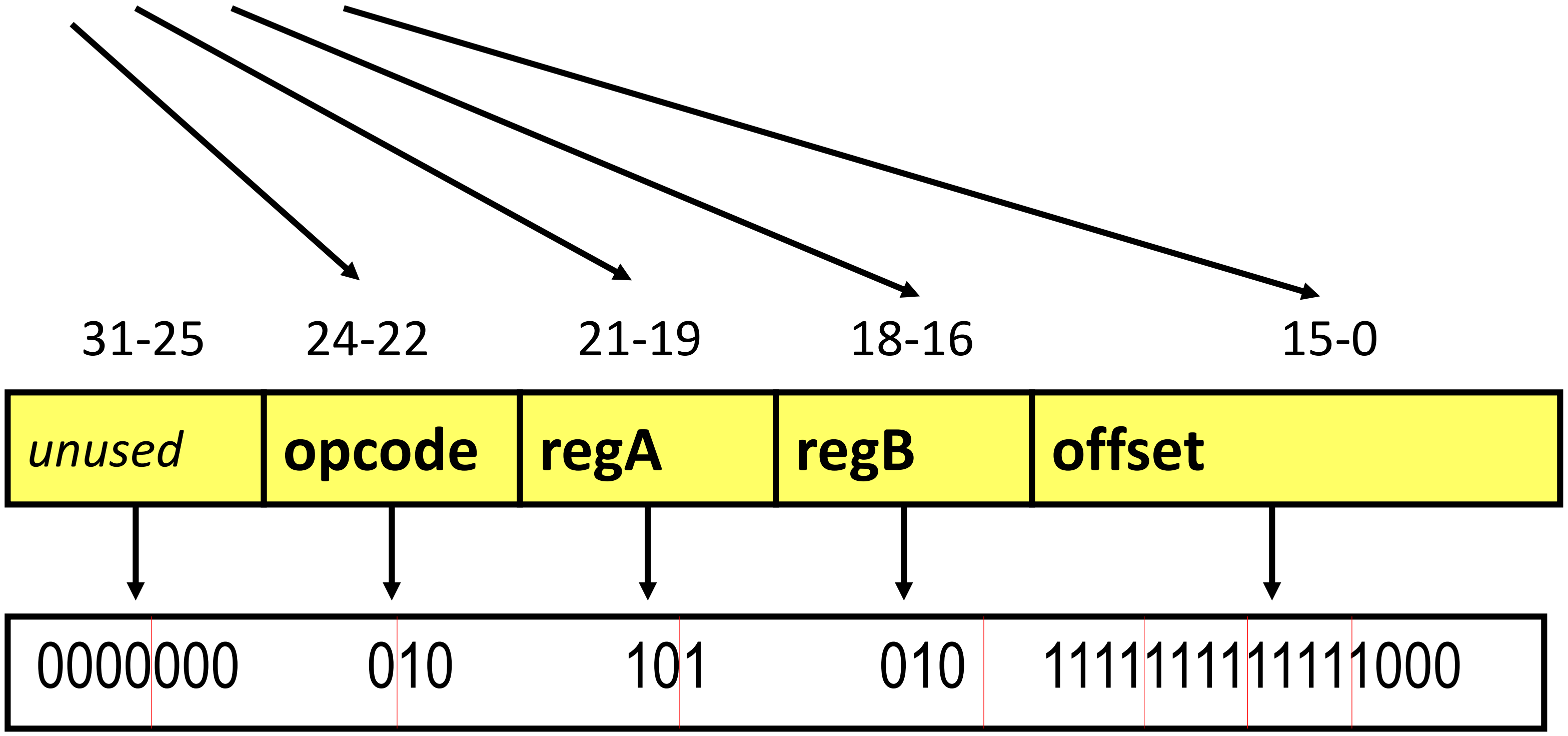
Convert to Hex → 0x005C0007

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

ARM Instruction Set

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

ARM ISA

ARM Arithmetic Instructions

- Format: three operand fields
  - Dest. register often the first one – check instruction format
  - e.g., add r3, r4, r7

- **C-code example: f = (g + h) – (i + j)**

r3→g
r4→h
r5→i
r6→j

```
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 Arithmetic Instructions – special 2nd operand

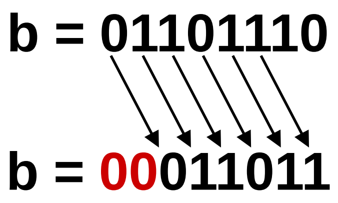
ARM ISA

- Format: second source operand can be a reg or imm (8 bit)
  - e.g., add r3, r4, #10
- **C-code example: f = g + 10**

```
add     r7, r5, #10
```

ARM Arithmetic Instructions – special 2nd operand

- Format: second operand can be a shift
- C/C++
  - a = b >> 2;
  - c = d << 4;



- ARM

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

ARM ISA

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)
  - Load constant from memory into a register, then use a register operand

- **C-code example: f = 0x104**

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

0x041 (0b000001000001)

↓

0x104 (0b000100000100)

ARM ISA

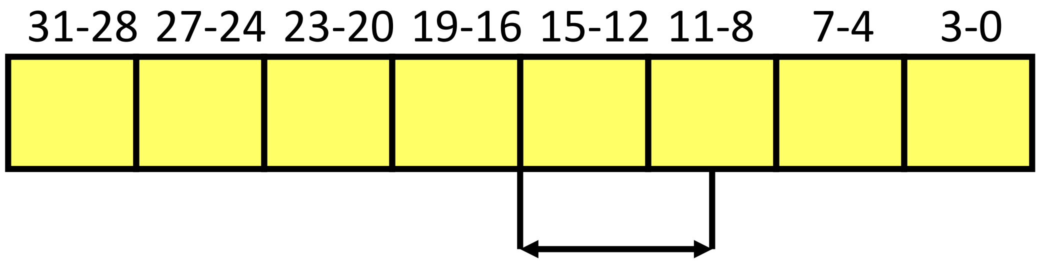
ARM Arithmetic Instructions - recap

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
mov	dest, src1, LSL src2imm	// logical left shift src1 by src2imm
mov	dest, src1, LSR src2imm	// logical right shift src1 by src2imm
cmp	src1, src2imm	// compare src1 to src2imm & set PSR

ARM ISA

Class Problem 2

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



Want these bits

Remember each hex digit is 4 bits

ARM ISA

ARM Memory Instructions

- Supports base + displacement and base + register modes only
  - Base is a register
  - Offset is a 12-bit immediate, either positive or negative

- Format: 2 registers (dest, base) and 12-bit immediate (offset)

  - Example:
    ldr   r3, [r4, #1000]   // load word
    Retrieves 32-bit value from memory location (r4 + 1000) and puts the result into r3

ARM ISA

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)

ARM ISA

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

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

  - Zero extend if the load is unsigned

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

ARM Memory Instructions - recap

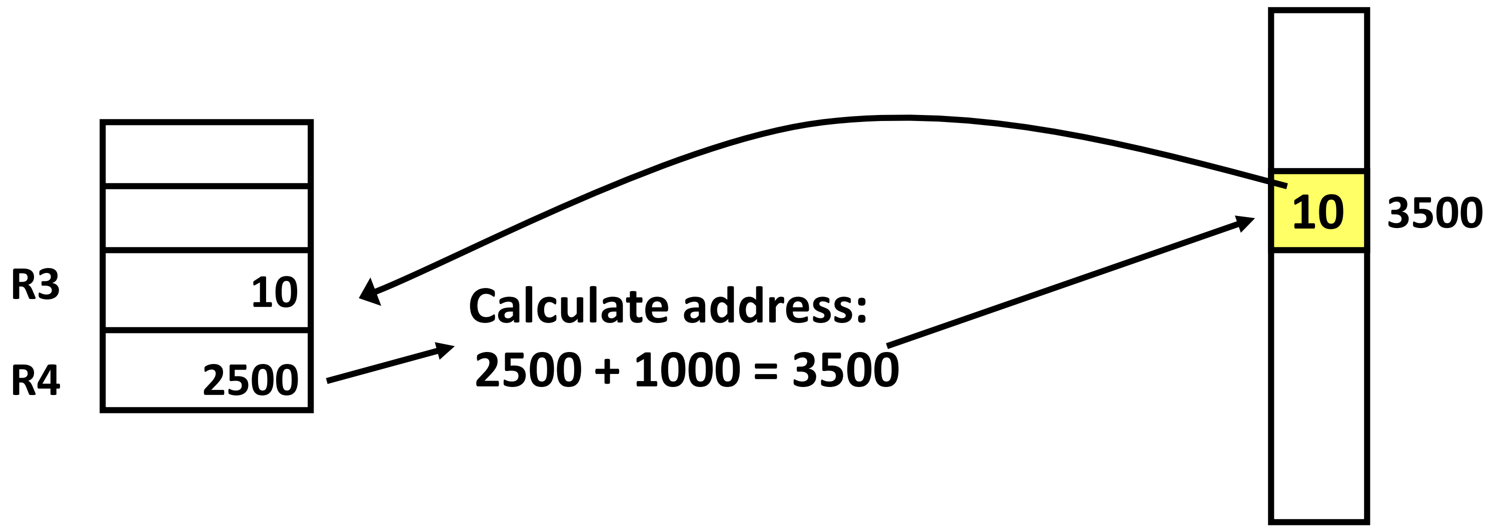
ARM ISA

- 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]

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)

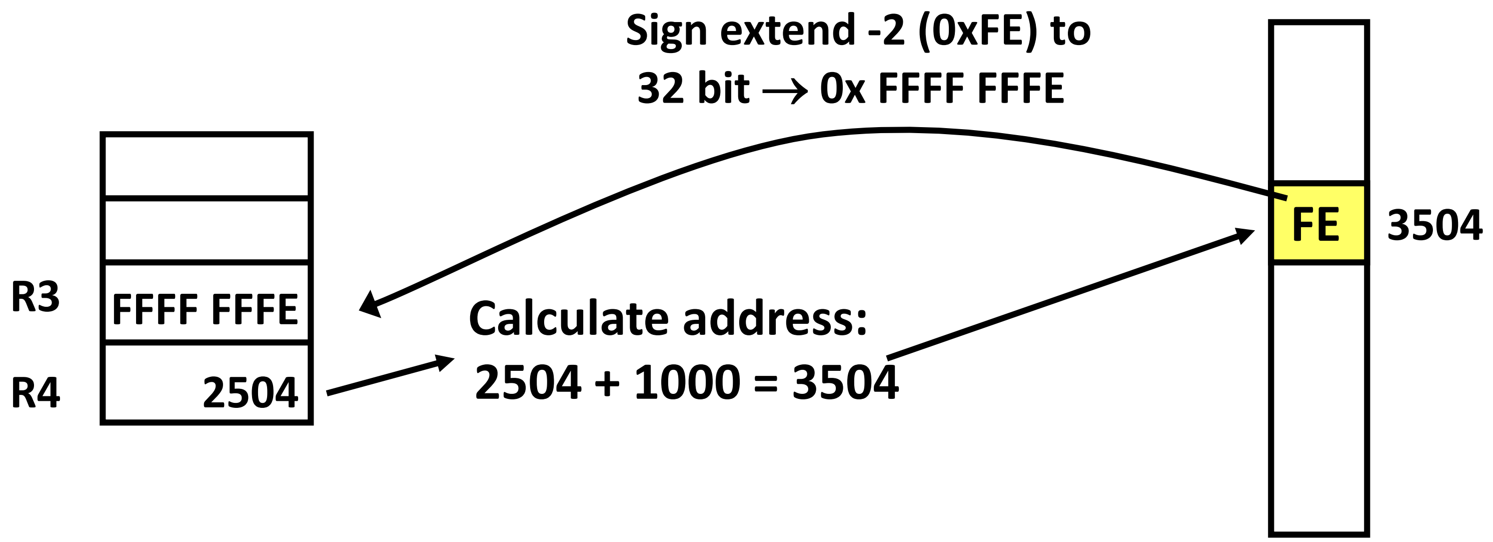


ARM ISA

Load Instruction in Action – other example

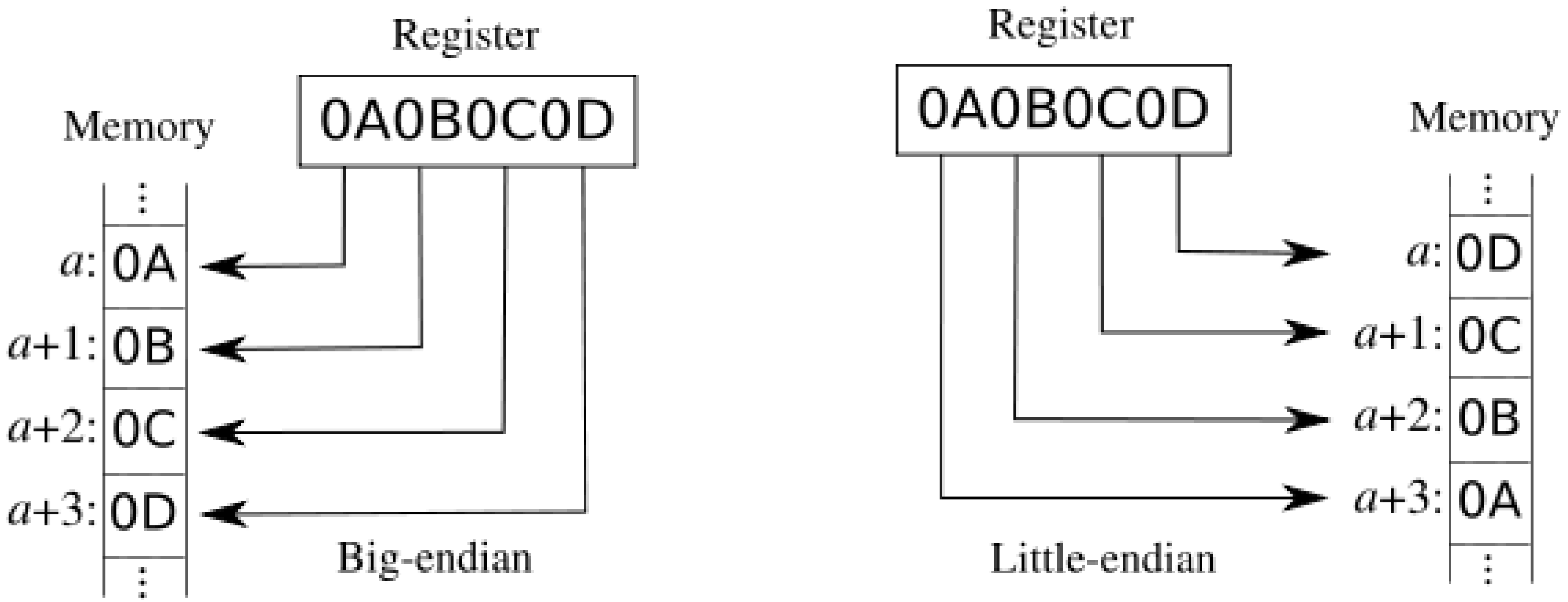
- ldrsb r3, [r4, #1000] // load signed byte

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


Sign extend -2 (0xFE) to 32 bit → 0x FFFF FFFE
R3
FFFF FFFE
R4
2504
Calculate address: 2504 + 1000 = 3504
FE
3504

Big Endian vs. Little Endian

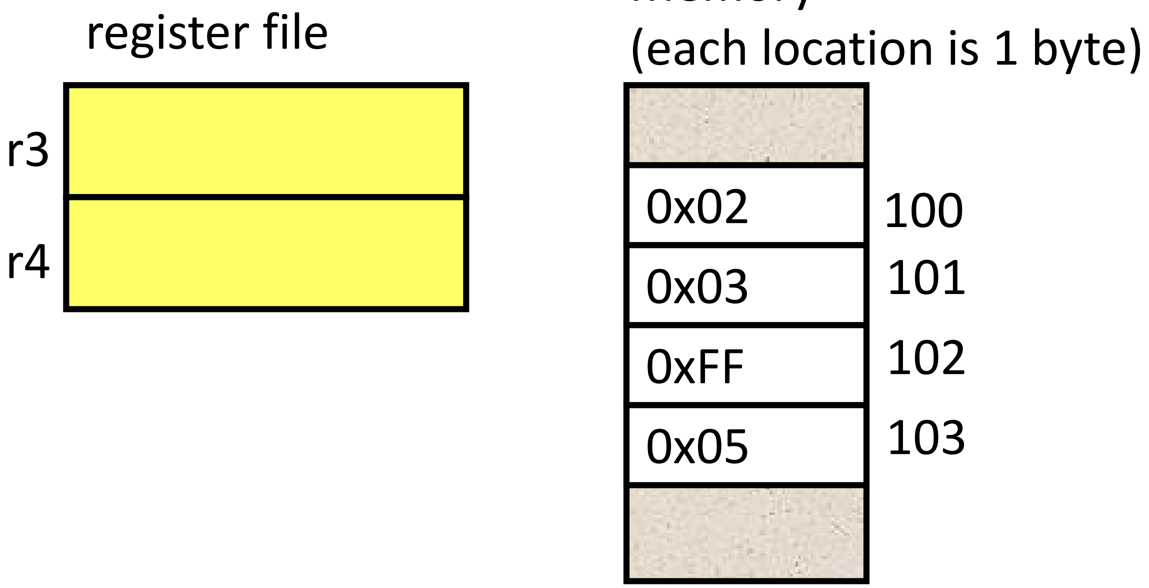
- Endian-ness: ordering of bytes within a word
  - Little - increasing numeric significance with increasing memory addresses
  - Big – The opposite, most significant byte first
  - The Internet is big endian, x86 and ARM are little endian



Example Code Sequence

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

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

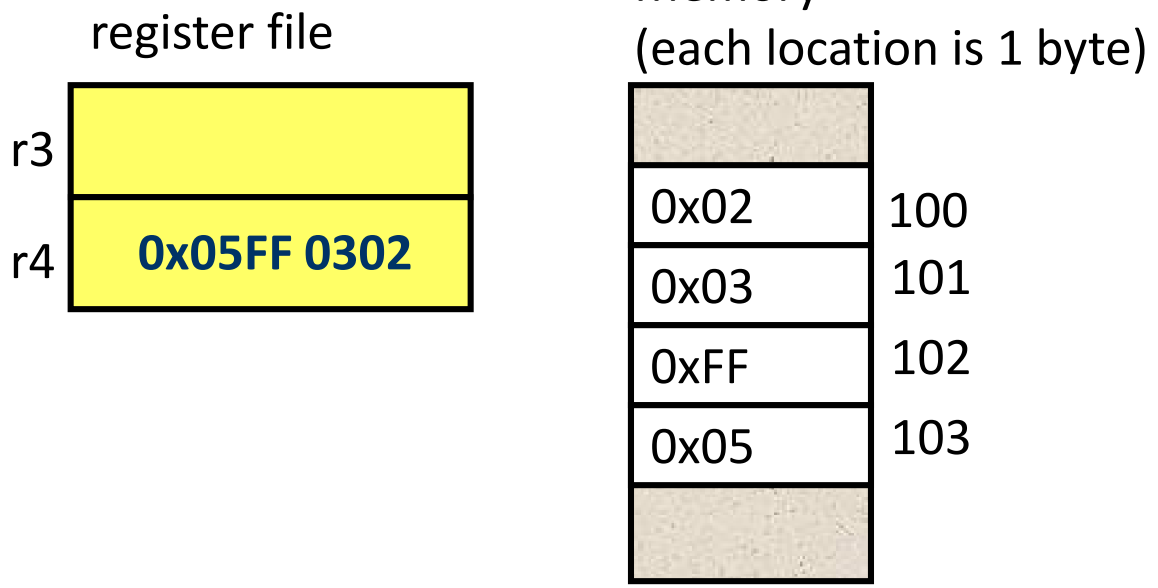


ARM ISA

Example Code Sequence – insn 1

What is the final state of memory once you execute the following instruction sequence?

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

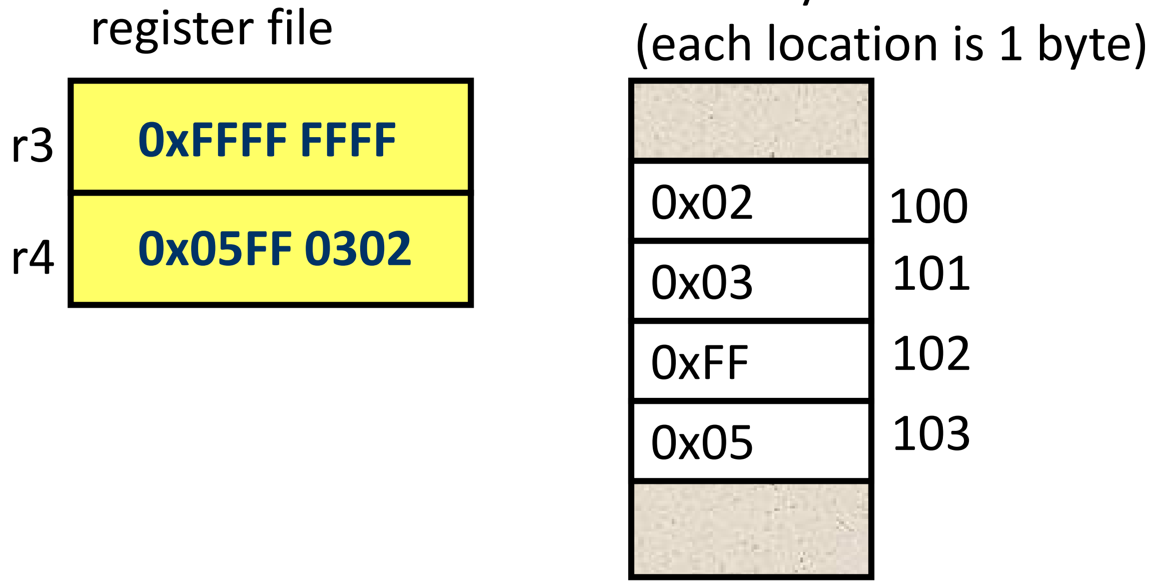


ARM ISA

Example Code Sequence – insn 2

What is the final state of memory once you execute the following instruction sequence?

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

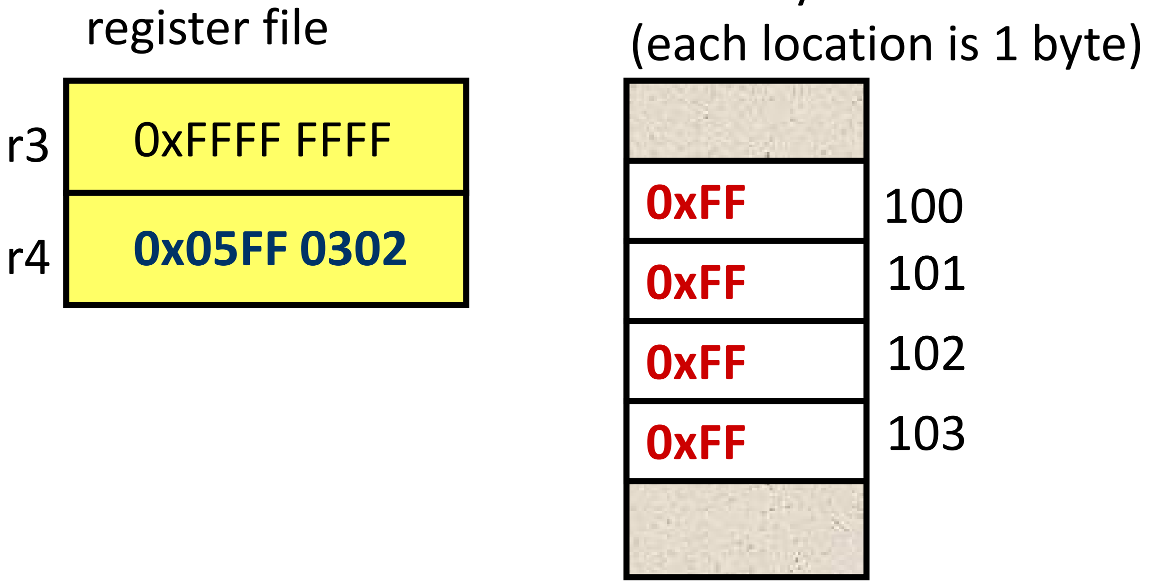


ARM ISA

Example Code Sequence – insn 3

What is the final state of memory once you execute the following instruction sequence?

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

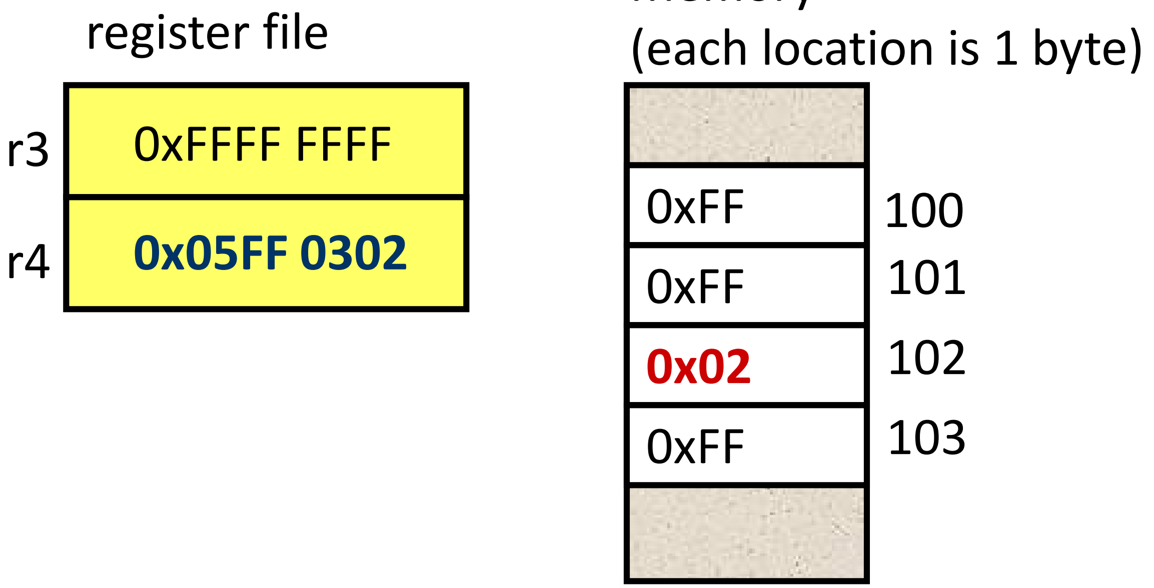


ARM ISA

Example Code Sequence – insn 4

What is the final state of memory once you execute the following instruction sequence?

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

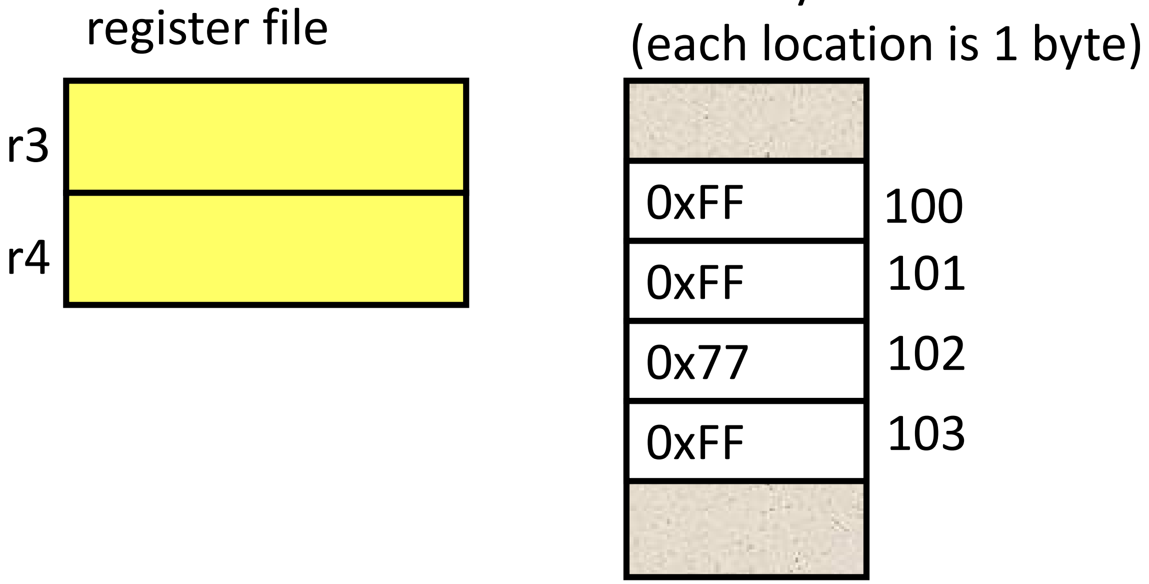


ARM ISA

Class Problem 3

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

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

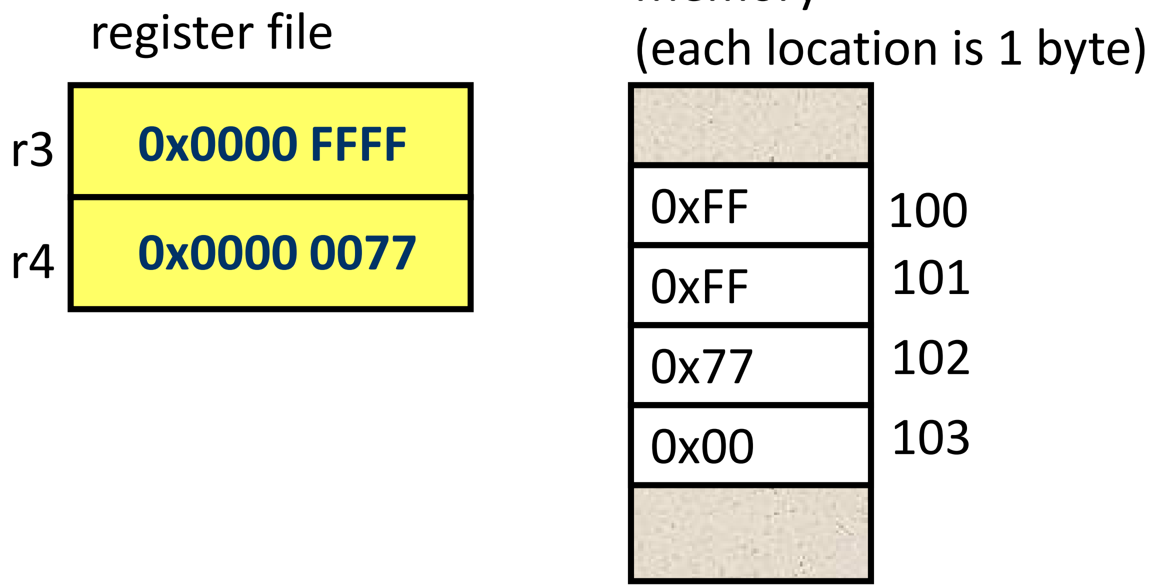


ARM ISA

Class Problem 3

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

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


register file
r3
0x0000 FFFF
r4
0x0000 0077
Memory
(each location is 1 byte)
0xFF
100
0xFF
101
0x77
102
0x00
103