

# EECS 370 - Lecture 15

## Exam Review



# Exam stuff

- Exam is on Thursday March 9<sup>th</sup> from 7-9pm.
  - Room assignments on Piazza (post [2068](#)).
- What can you have?
  - We'll supply the ARMv8 quick reference card and a handout
    - See post [2068](#) on Piazza for the handout
  - You can bring an 8.5 by 11 sheet with notes (both sides fine)
  - Calculator without wireless capability
    - No storing stuff in it.
- Topics?
  - Through lecture 15
    - Again, see the Piazza post for details.
  - HW 1-3
  - Programming assignments 1, 2a, 2l.

# Exam Advice

- Read the first page of the midterm exam
- Sign the honor pledge!
  - No signature = no grade
- Pay attention to time
  - Don't get stuck on a single problem
  - Answer as many questions as possible



# Preparing for the Exam

- Review the lecture notes
- Go over old homework assignments
- Go over old discussion examples
- Skim through the book sections specified
- Past exams w/solutions will be posted to course website
  - Complete without looking at the solutions!
  - Some topics are no longer relevant (e.g., MIPS ARMv7)



# Important Topics

- Representing values in hardware
  - Binary, octal, hexadecimal conversions
  - 2's complement representation
  - Floating point formats
- Instruction sets
  - Assembly code LEGv8/ LC2k: write & understand
  - Converting to machine code
  - Addressing modes for load/store instructions
  - Conditional instructions



# Important Topics

- Converting C to assembly and back
  - Data alignment
  - Basic statements
  - Control flow constructs
- Running programs
  - Data organization (stack, heap, static, text)
  - Stack frames, stack and frame pointers
  - Object files: symbol table and relocation table
  - Caller/callee-saved registers
  - Compiler, linker, loader



# Important Topics

- Logic gates, devices and basic state machines
  - AND, OR, NOT, XOR, etc. gates
  - Decoders, MUXes, ALUs, etc.
  - Latches and flip-flops
  - Next state and output logic (via control ROM)
- Data path
  - Single cycle, multi-cycle, pipeline
  - CPI and performance computations
  - Adding new instructions



## Review Questions

If a topic is not covered in  
this review, it does not  
imply that it is not  
important!





# Memory Layout

- How many bytes does the C data structure require (assuming a 64-bit machine)?

```
struct foo {  
    double *w;  
    char x;  
    int y;  
    char z[10];  
};
```

←8 bytes

←1 byte

←4 bytes + 3 bytes for alignment

←1 \* 10 bytes

←26 bytes + 6 for padding= 32 bytes

- How could this structure be rewritten to reduce memory usage?



# Memory Layout

- How many bytes does the C data structure require?

```
struct foo {  
    double *w;    ← 8 bytes  
    char x;       ← 1 byte  
    int y;        ← 4 bytes + 3 bytes for alignment  
    char z[10];   ← 1 * 10 bytes  
};               ← 26 bytes + 6 bytes padding = 32
```

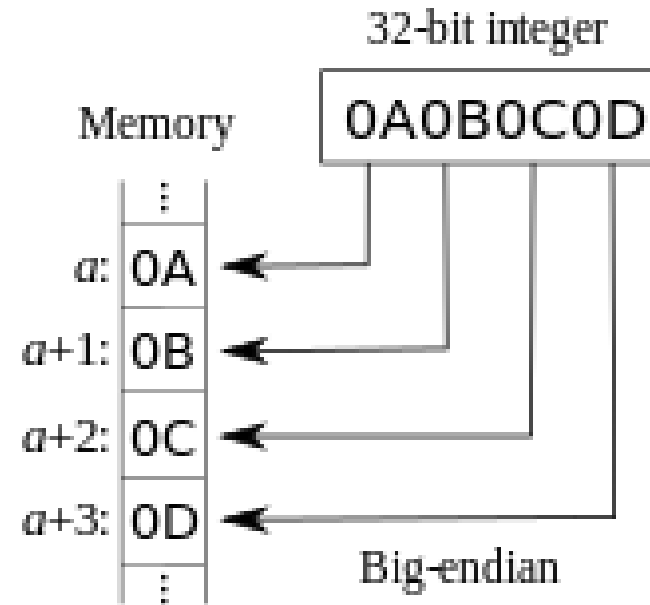
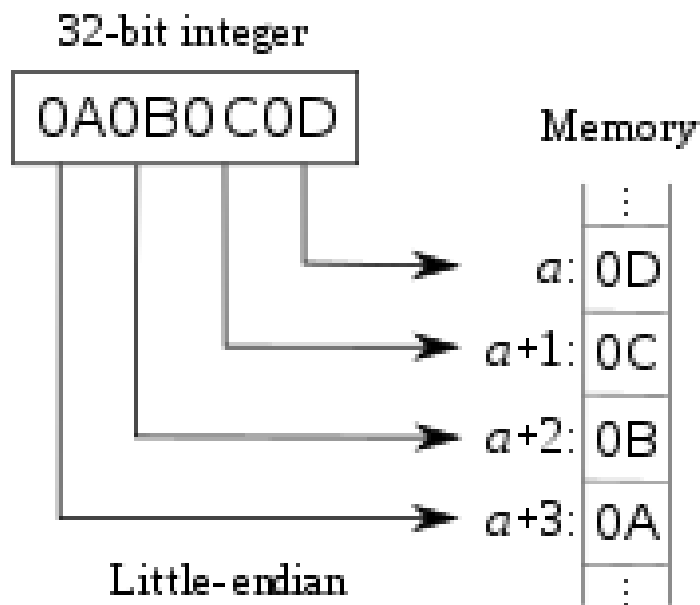
- How could this structure be rewritten to reduce memory usage?

```
struct foo {  
    double *w;    ← 8 bytes  
    int y;        ← 4 bytes  
    char z[10];   ← 1 * 10 bytes  
    char x;       ← 1 byte  
};               ← 23 bytes + 1 byte padding = 24
```



# 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 is little endian, LEG and ARMv8 can switch



# Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR    X3, [X5, #100]
STURB   X4, [X5, #102]
```

register file



Memory  
(each location is 1 byte)

0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

little endian

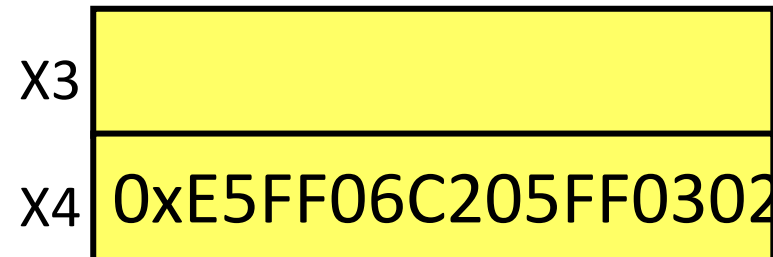
*We shown the registers as blank. What do they actually contain before we run the snippet of code*

# Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR      X4, [X5, #100]
LDURB     X3, [X5, #102]
STUR      X3, [X5, #100]
STURB     X4, [X5, #102]
```

register file



Memory

(each location is 1 byte)

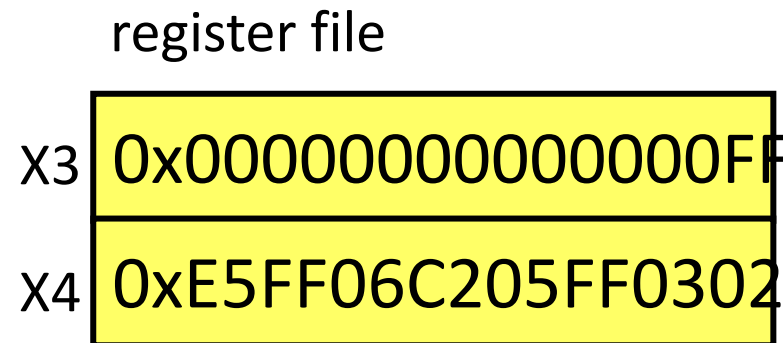
0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

little endian

# Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR    X3, [X5, #100]
STURB   X4, [X5, #102]
```



Memory  
(each location is 1 byte)

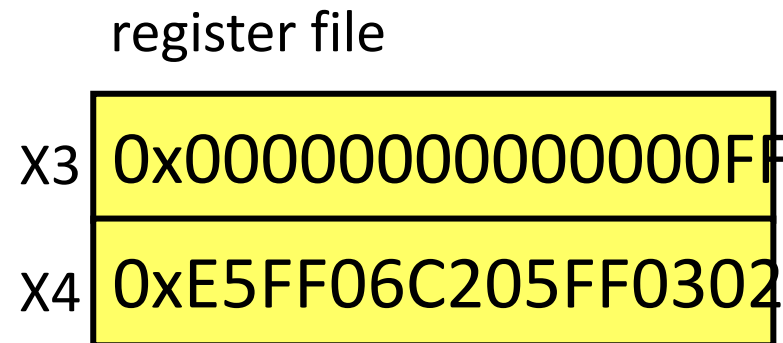
0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

little endian

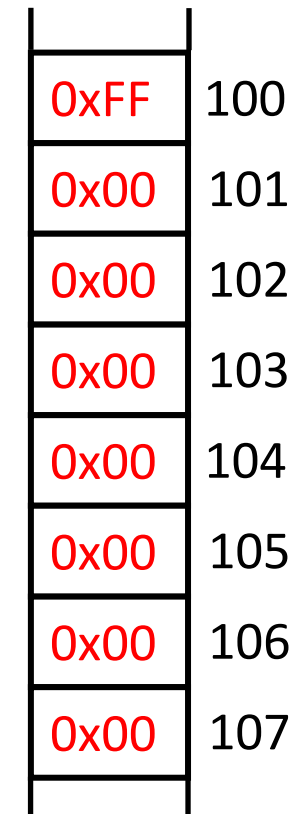
# Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR    X3, [X5, #100]
STURB   X4, [X5, #102]
```



Memory  
(each location is 1 byte)



little endian

# Example Code Sequence

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR    X3, [X5, #100]
STURB   X4, [X5, #102]
```

register file

X3	0x0000000000000000FF
X4	0xE5FF06C205FF0302

Memory  
(each location is 1 byte)

0xFF	100
0x00	101
0x02	102
0x00	103
0x00	104
0x00	105
0x00	106
0x00	107

little endian



## Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR    X4, [X5, #100]
LDURB   X3, [X5, #102]
STUR    X3, [X5, #100]
LDURSW  X4, [X5, #102]
```

register file



Memory  
(each location is 1 byte)

0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

little endian

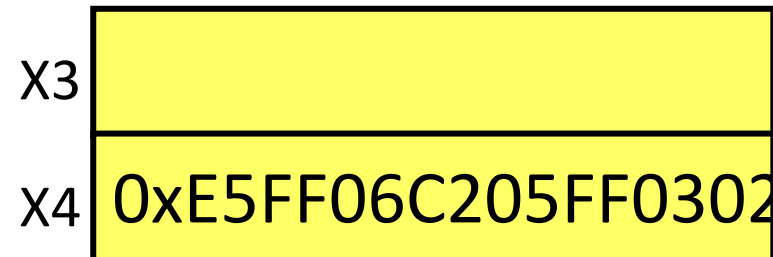
*We shown the registers as blank. What do they actually contain before we run the snippet of code*

## Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR      X4, [X5, #100]
LDURB     X3, [X5, #102]
STUR      X3, [X5, #100]
LDURSW    X4, [X5, #102]
```

register file



Memory

(each location is 1 byte)

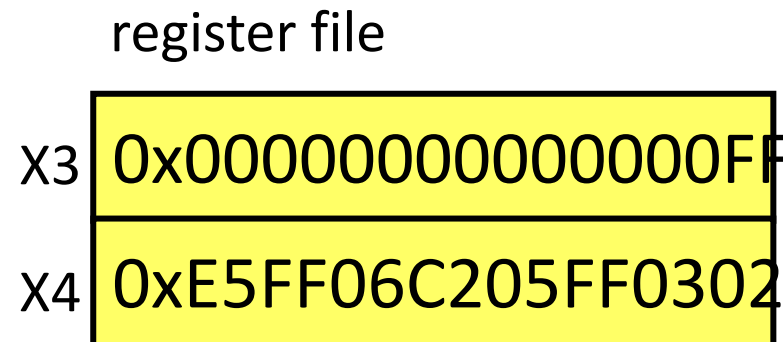
0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

little endian

## Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR      X4, [X5, #100]
LDURB     X3, [X5, #102]
STUR      X3, [X5, #100]
LDURSW    X4, [X5, #102]
```



Memory  
(each location is 1 byte)

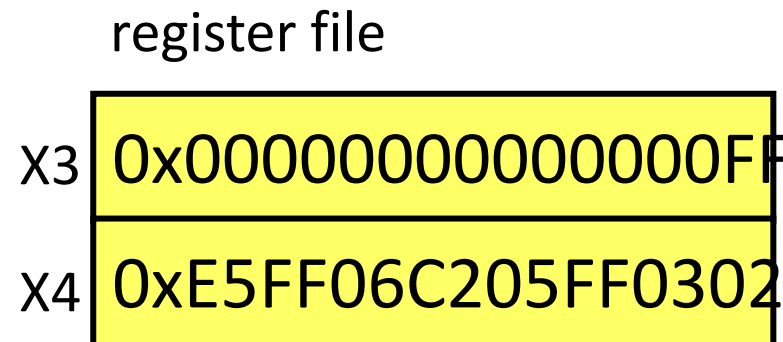
0x02	100
0x03	101
0xFF	102
0x05	103
0xC2	104
0x06	105
0xFF	106
0xE5	107

little endian

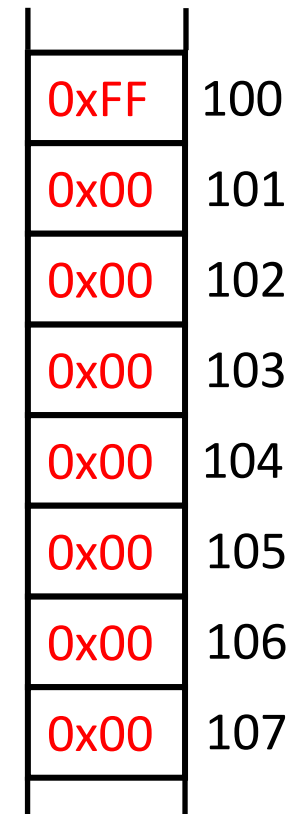
## Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR      X4, [X5, #100]
LDURB     X3, [X5, #102]
STUR      X3, [X5, #100]
LDURSW    X4, [X5, #102]
```



Memory  
(each location is 1 byte)

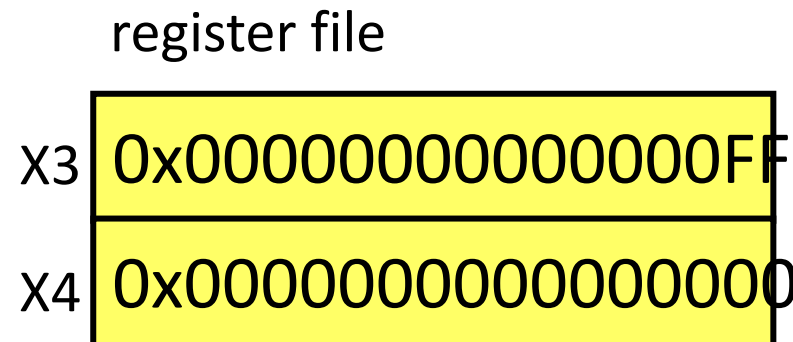


little endian

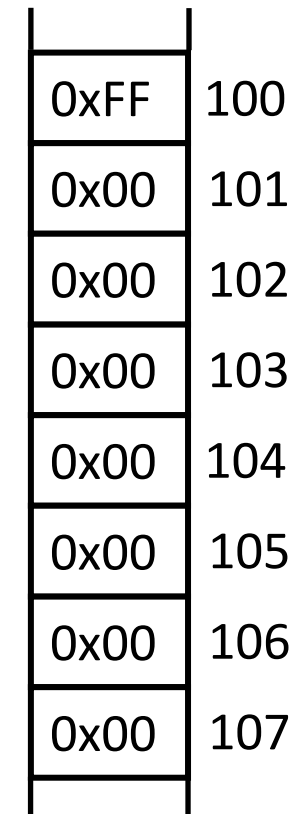
## Example Code Sequence (2)

What is the final state of memory once you execute the following instruction sequence? (assume X5 has the value of 0)

```
LDUR      X4, [X5, #100]
LDURB     X3, [X5, #102]
STUR      X3, [X5, #100]
LDURSW    X4, [X5, #102]
```



Memory  
(each location is 1 byte)



little endian

## Load Instruction Sizes

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

- LDUR X3, [X4, #1000]
  - Load (unscaled) to register—retrieve a double word (64 bits) from address (X4+1000)
- LDURH X3, [X4, #1000]
  - Load halfword (16 bits) from address (X4+1000) to the low 16 bit in X3—top 48 bits of X3 are set zero
- LDURB X3, [X4, #1000]
  - Load byte (8 bits) from address (X4+1000) and put in the low 8 bits of X3—zero extend the destination register X3 (top 56 bits)
- What about loading words?
- LDURSW X3, [X4, #1000]
  - retrieve a word (32 bits) from address (X4+1000) and put in lower half of X3—top 32 bits of X3 are sign extended
  - the most significant bit of the word at address (X4+1000) is copied into the top 32 bits of X3

# LEG Pseudos

MOV X3, X4

- Expanded to:  
LSL X3, X4, #0 // many other possibilities

CMP X9, X10 // compare X9 to X10 and set condition code

- Expands to:  
SUBS XZR, X9, X10 // use X9 – X10 to set condition codes
  - We allow writes to XZR (X31) to set condition codes—the result is discarded
  - Immediates allowed



# IEEE Floating point format (single precision)

- Sign bit: (0 is positive, 1 is negative)
- Significand: (also called the *mantissa*; stores the 23 most significant bits after the decimal point)
- Exponent: used biased base 127 encoding
  - Add 127 to the value of the exponent to encode:
    - -127  $\rightarrow$  00000000      1  $\rightarrow$  10000000
    - -126  $\rightarrow$  00000001      2  $\rightarrow$  10000001
    - ...                                  ...
    - 0  $\rightarrow$  01111111      128  $\rightarrow$  11111111
- How do you represent zero ? Special convention:
  - Exponent: -127 (all zeroes ), Significand 0 (all zeroes), Sign + or -





# IEEE 754 Floating Point

- What is the value, in binary, of the following IEEE 754-encoded floating-point number?
- What is the value, in decimal, for the same number?

0	01111111	101000000000000000000000
---	----------	--------------------------

$1.101 \times 2^0$

$1.625 \times 2^0$



# Symbol Table & Relocation Table

File main.c

```
1: int r;  
2: extern int x;  
3: extern void foobar();  
4: void main(int a) {  
5:     reference to x  
6:     reference to r  
7:     foobar();  
8: return; }
```

File foobar.c

```
1: int x;  
2: int y;  
3: void foobar() {  
4:     int t;  
5:     reference to x  
6:     reference to y  
7:     reference to t  
8: return; }
```

What symbols appear in the symbol tables?

What instructions appear in the relocation tables?

**Symbol Table:**

**r, x, foobar, main**

**Relocation Table:**

**5, 6, 7**

**Symbol Table:**

**x, y, foobar**

**Relocation Table:**

**5, 6**



# Assigning Variables to Memory Spaces

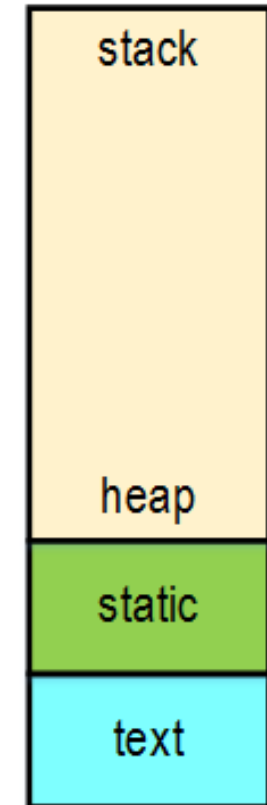
```
int z;  
static float q;  
int foo() {  
    int a, b, c;  
    static int d;  
    a = 1;  
    b = bar(a);  
    c = a + b;  
    char *s;  
  
    s = malloc(10);  
    printf(s);  
    return;  
}
```

**z = data**  
**q = data**

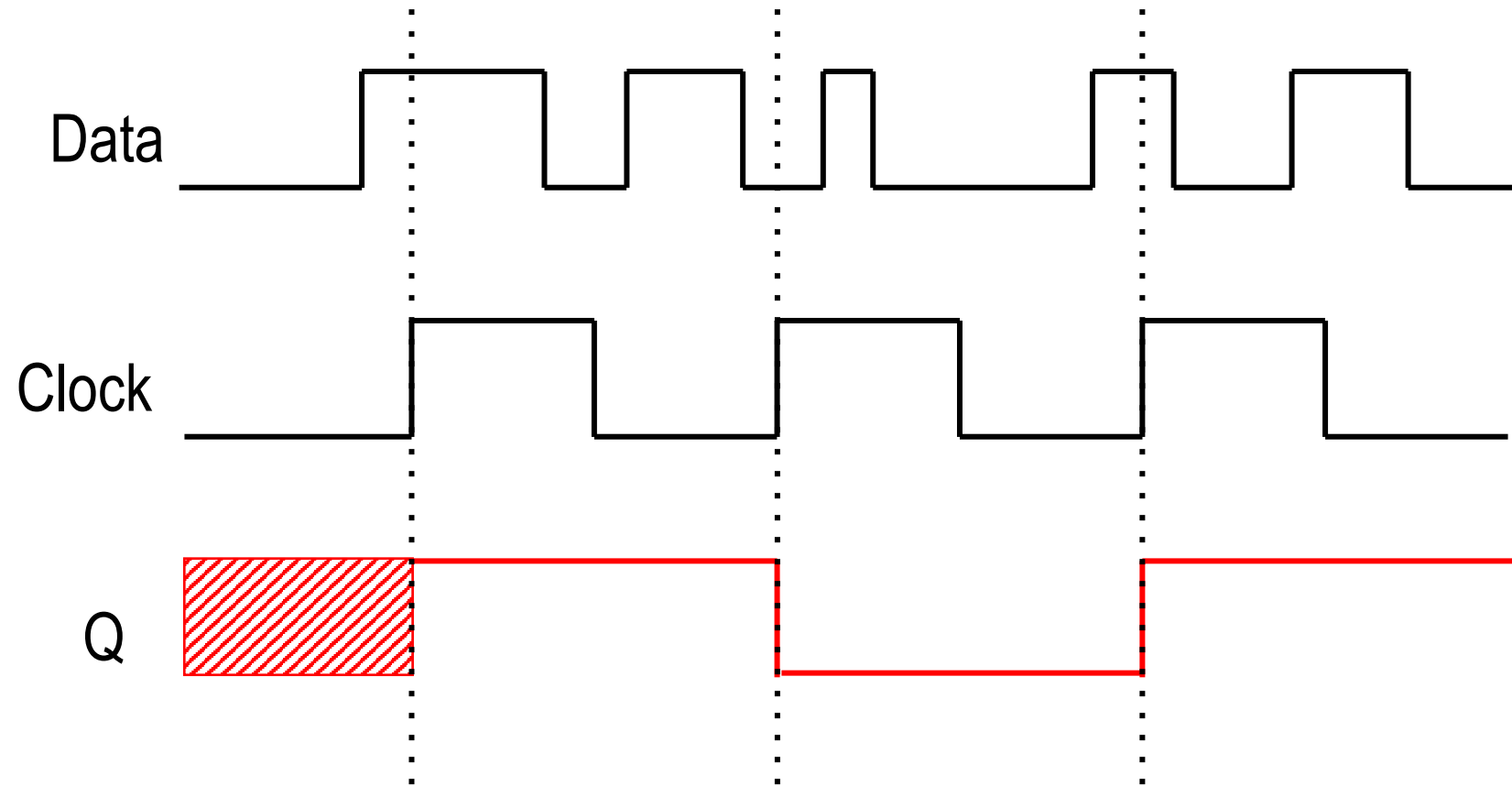
**a, b, c = stack**  
**d = data**

**s = stack;**  
**t = stack,**

**\*s = heap**  
**s = stack**



## Q for a D flip-flop?



# Datapath

- We've covered three datapaths
  - Single-cycle
  - Multi-cycle
  - Pipeline



# Data path Performance Questions

- Consider an LC2K program which consists of:
  - 20 loads
  - 25 stores
  - 15 beqs
  - 10 nors
  - 30 adds
- What would be the execution time for:
  - Our single-cycle datapath with a clock period of 100ns
  - Our multi-cycle datapath with a clock period of 20ns
  - Our pipelined datapath with a clock period of 25ns (with no stalls).

10,000 ns (10  $\mu$ s)

8,400 ns (8.4  $\mu$ s)

2,600 ns (2.6  $\mu$ s)



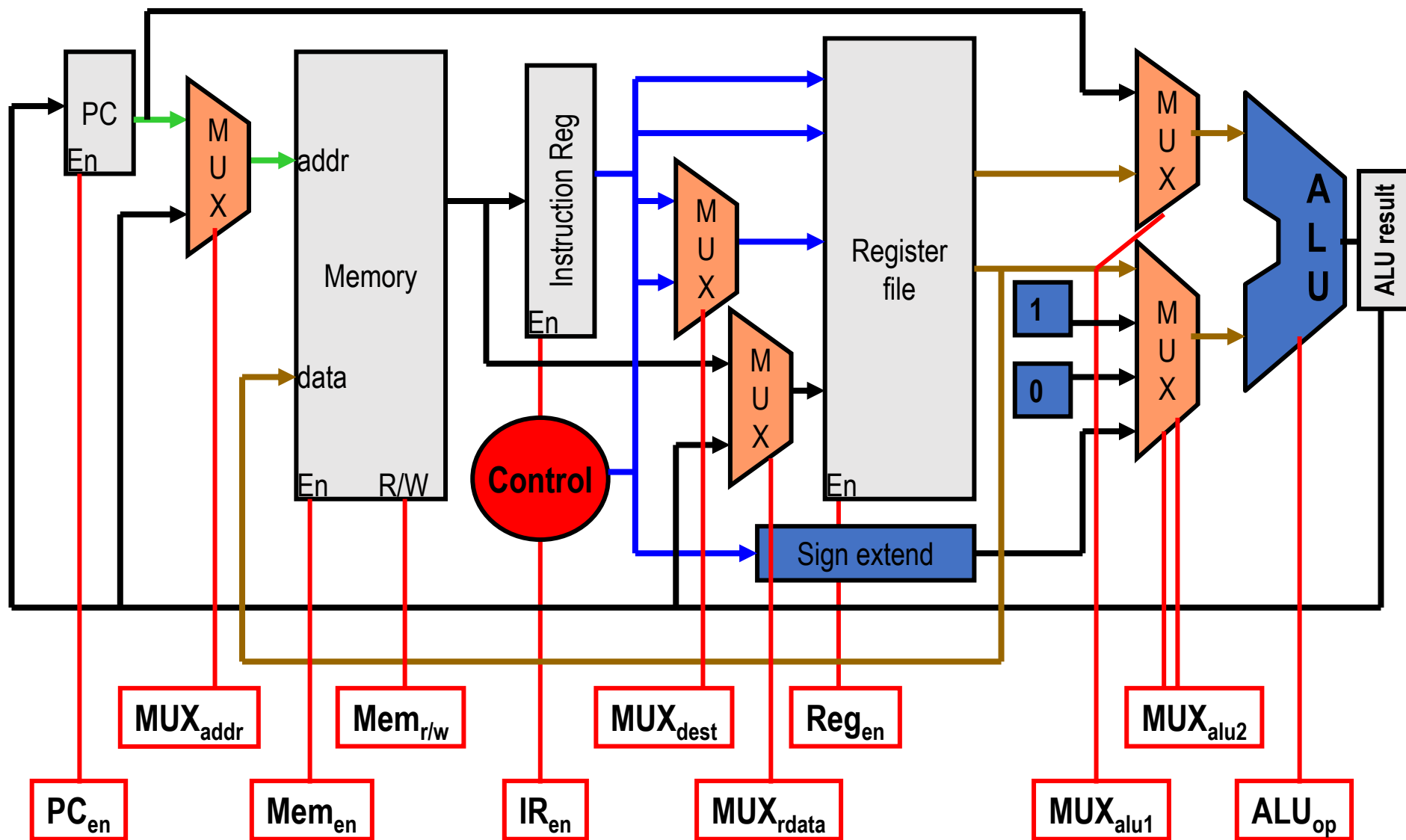
# Multicycle Datapath

Consider the above LC-2K multicycle datapath covered in lecture. We want to provide hardware support for a new instruction:

$\text{destReg} = \text{regA}++ + \text{regB}$

1. List any new hardware components that need to be introduced to the LC-2K datapath to directly support the new instruction.
2. List any new control signals that need to be introduced to the LC-2K control to provide support for the new instruction.







# Multicycle Datapath – F05E2 (Q. II.C cont.)

Give a cycle by cycle description of the LC-2K operation when executing the new instruction. For each cycle, give the following information: Single-sentence description of what the cycle is about, and Register updates. Use as few cycles as possible.

Cycle 1	Fetch instruction Instruction Register = new instruction ALU Result = PC + 1
Cycle 2	Decode instruction and read registers PC = PC + 1
Finish the rest	



# Multicycle Datapath – F05E2 (Q. II.C cont.)

Cycle 1	Fetch instruction Instruction Register = new instruction ALU Result = PC + 1
Cycle 2	Decode instruction and read registers PC = PC + 1



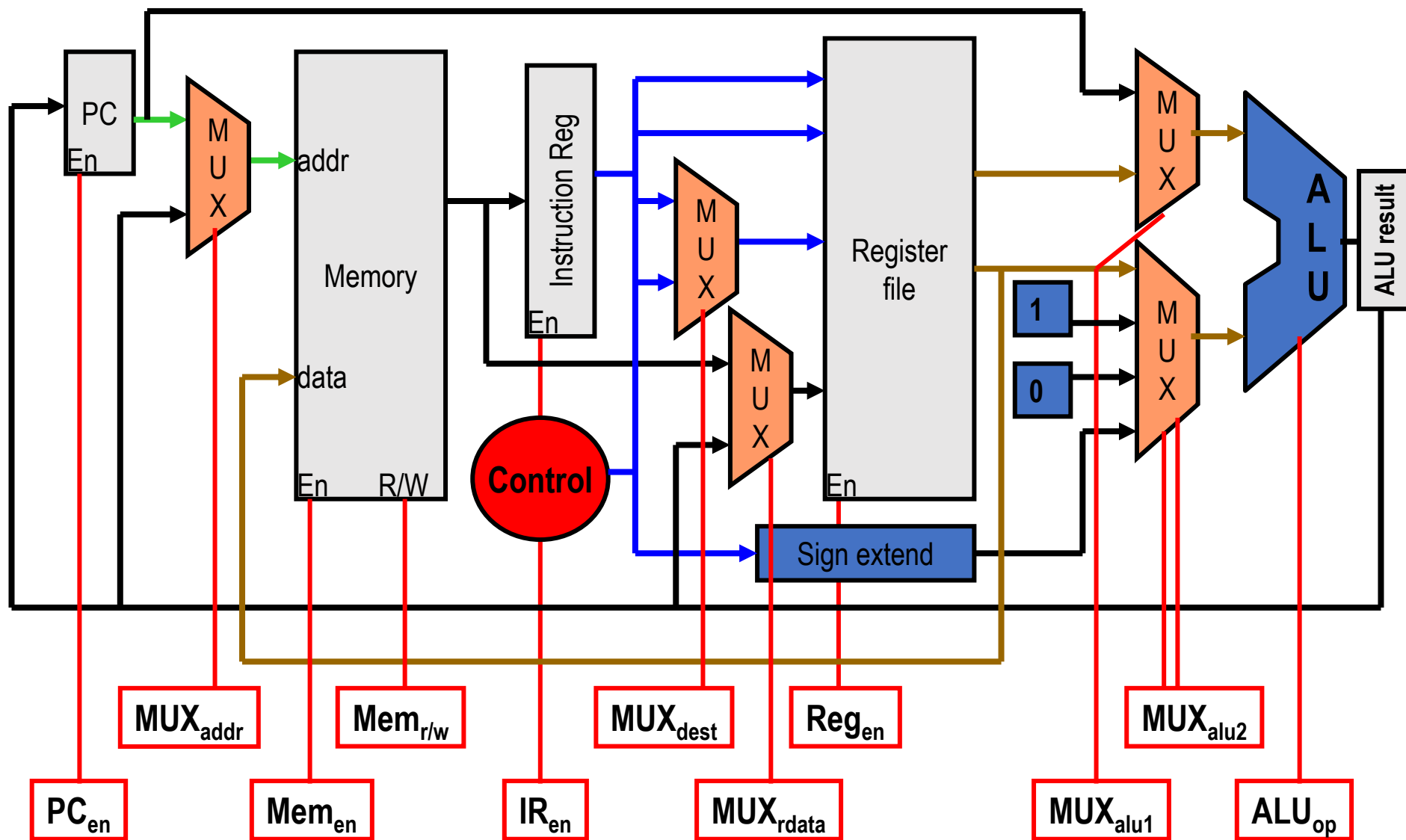
destReg = regA++ + regB →

destReg = regA++ + regB  
regA = regA + 1

## Question II.C 3 - answer

Cycle 1	// Fetch instruction Instruction Register = new instruction ALU Result = PC + 1
Cycle 2	// Decode instruction and read registers PC = PC + 1
Cycle 3	// Add regA and regB ALUResult = regA + regB
Cycle 4	// Store ALUResult to destR RegisterFile[destR] = ALUResult // Add regA and 1 ALUResult = regA + 1
Cycle 5	// Store ALUResult to regA <b>RegisterFile[regA] = ALUResult</b>





# Question II.C 1 and 2 - answers

1. Add a entry to the MUX connecting to the register write address port to the register file, making it 3-1 MUX. Connect the instruction\_reg[21:19] (the regA field) to the new MUX input.
2. A second control bit is required for the new MUX.

# Pipelining

- Say you have the following LC2K program:
  - beq 1 0 0                   //not taken
  - add 1 1 1
  - lw 1 2 15
- How long would it take for that program to finish the lw on our 5-stage pipeline with “detect and stall” used to resolve data and control hazards?

Cycle	1	2	3	4	5	6	7	8	9	10	11	12

# Pipelining

- Say you have the following LC2K program:
  - beq 1 0 0                   //not taken
  - add 1 1 1
  - lw 1 2 15
- How long would it take for that program to finish the lw on our 5-stage pipeline with “detect and stall” used to resolve data and control hazards?

Cycle	1	2	3	4	5	6	7	8	9	10	11	12
beq	F	D	X	M	W							
add					F	D	X	M	W			
lw						F	D*	D*	D	X	M	W