CS2100 Tutorial

Control

will start at :05 as usual

Slides by Theodore, adapted from Prof. Aaron and Prof. Anandha's slides

Assignment 1

- I marked Q1+4+Admin; any questions ask after tutorial
    - Common mistakes:
        - using pow()
        - naming files wrongly
        - failing testcase (can't help for this)
- Please check Q5
    - NA and 00 should be accepted for the relevant questions
    - Still check in case of spurious errors

Midterms

- All the best :)
- Open book, bring your "cheatsheet"
  - MIPS Sheet
  - Control signal from lecture / tutorial (?)
  - Anything else important to you
  - (Sometimes making your own "cheatsheet" helps with revision
- Bring calculator

From lecture slide:

Generating ALUControl Signal

Opcode	ALUop	Instruction Operation	Funct field	ALU action	ALU control
lw	00	load word	XXXXXX	add	0010
sw	00	store word	XXXXXX	add	0010
beq	01	branch equal	XXXXXX	subtract	0110
R-type	10	add	10 0000	add	0010
R-type	10	subtract	10 0010	subtract	0110
R-type	10	AND	10 0100	AND	0000
R-type	10	OR	10 0101	OR	0001
R-type	10	set on less than	10 1010	set on less than	0111

Instruction Type	ALUop
lw / sw	00
beq	01
R-type	10

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

Generation of 2-bit **ALUop** signal will be discussed later

From lecture slide:

Design of ALU Control Unit (1/2)

- Input: 6-bit **Funct** field and 2-bit **ALUop**
- Output: 4-bit **ALUcontrol**
- Find the simplified expressions

ALUcontrol3 = 0

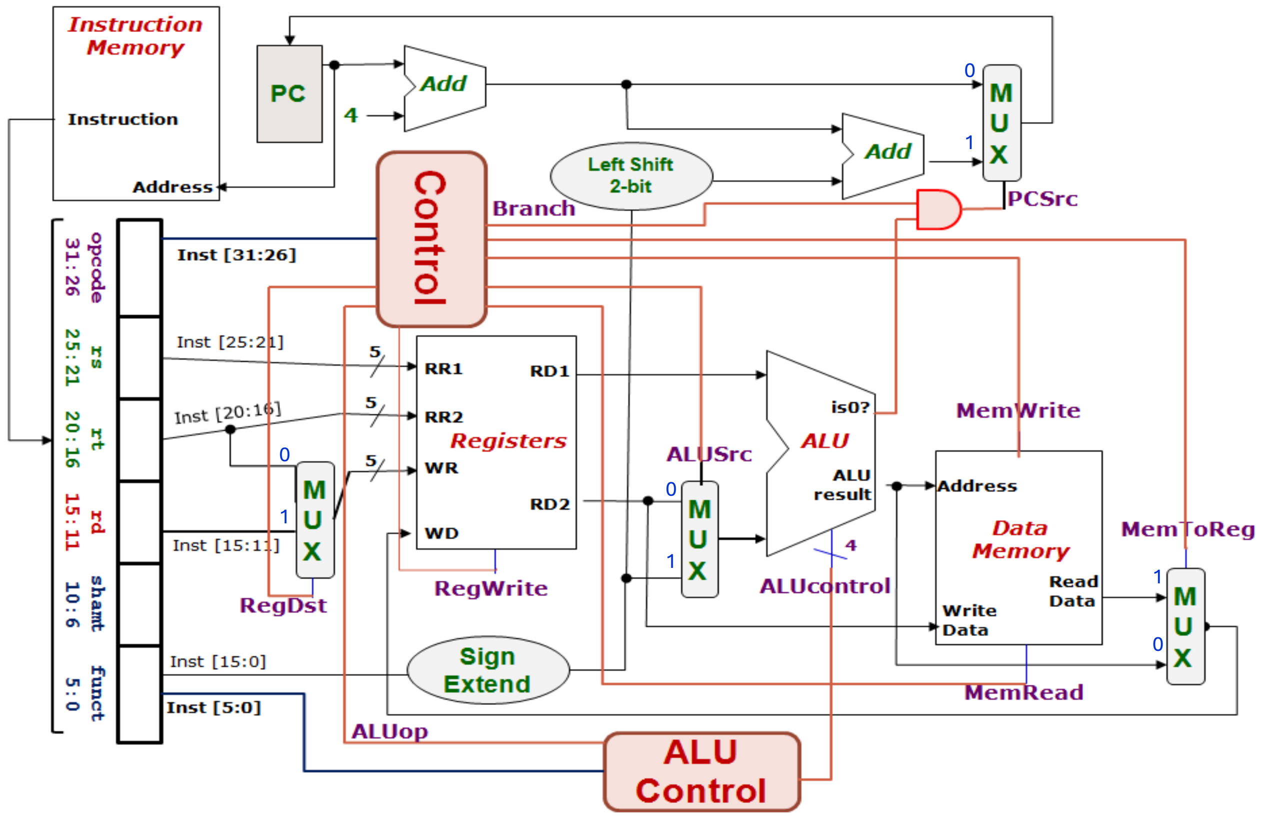
ALUcontrol2 = ALUop0 + ALUop1· F1

	ALUop		Funct Field (F[5:0] == Inst[5:0])						ALU control
	MSB	LSB	F5	F4	F3	F2	F1	F0	
lw	0	0	X	X	X	X	X	X	0 0 1 0
sw	0	0	X	X	X	X	X	X	0 0 1 0
beq	~~0~~ X	1	X	X	X	X	X	X	0 1 1 0
add	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	0	0	0 0 1 0
sub	1	~~0~~ X	~~1~~ X	~~0~~ X	0	0	1	0	0 1 1 0
and	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	0	0 0 0 0
or	1	~~0~~ X	~~1~~ X	~~0~~ X	0	1	0	1	0 0 0 1
slt	1	~~0~~ X	~~1~~ X	~~0~~ X	1	0	1	0	0 1 1 1

From lecture slide:

Control Design: **Outputs**

	RegDst	ALUSrc	MemTo Reg	Reg Write	Mem Read	Mem Write	Branch	ALUop	
								op1	op0
R-type	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1



```
lw $24, 0($15)
```

R[rt] = M[R[rs]+SignExtImm]

100011 01111 11000 0000000000000000

Q1(b)

Red means the data is actually used.

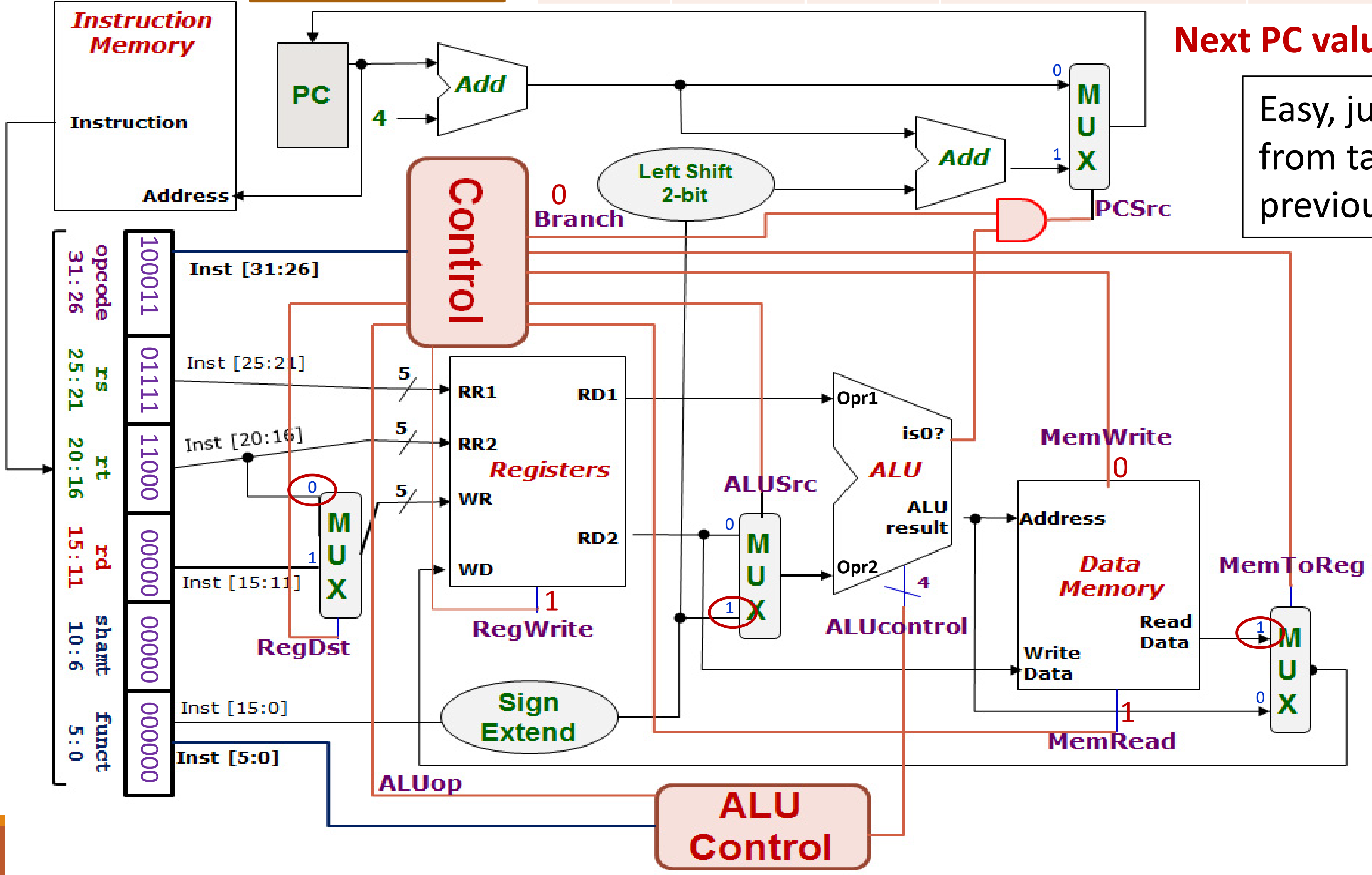
Blue means the data is not used.

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$15	$24	$24	MEM([$15]+0)	[$15]	0	[$15]+0	[$24]

Next PC value = PC + 4

Easy, just copy from tables in previous slides.

RegDest	0
RegWrite	1
ALUSrc	1
MemRead	1
MemWrite	0
MemToReg	1
Branch	0
ALUop	00
ALUcontrol	0010



`beq $1, $3, 12`

If (R[rs]==R[rt]) PC=PC+4+BrAddr

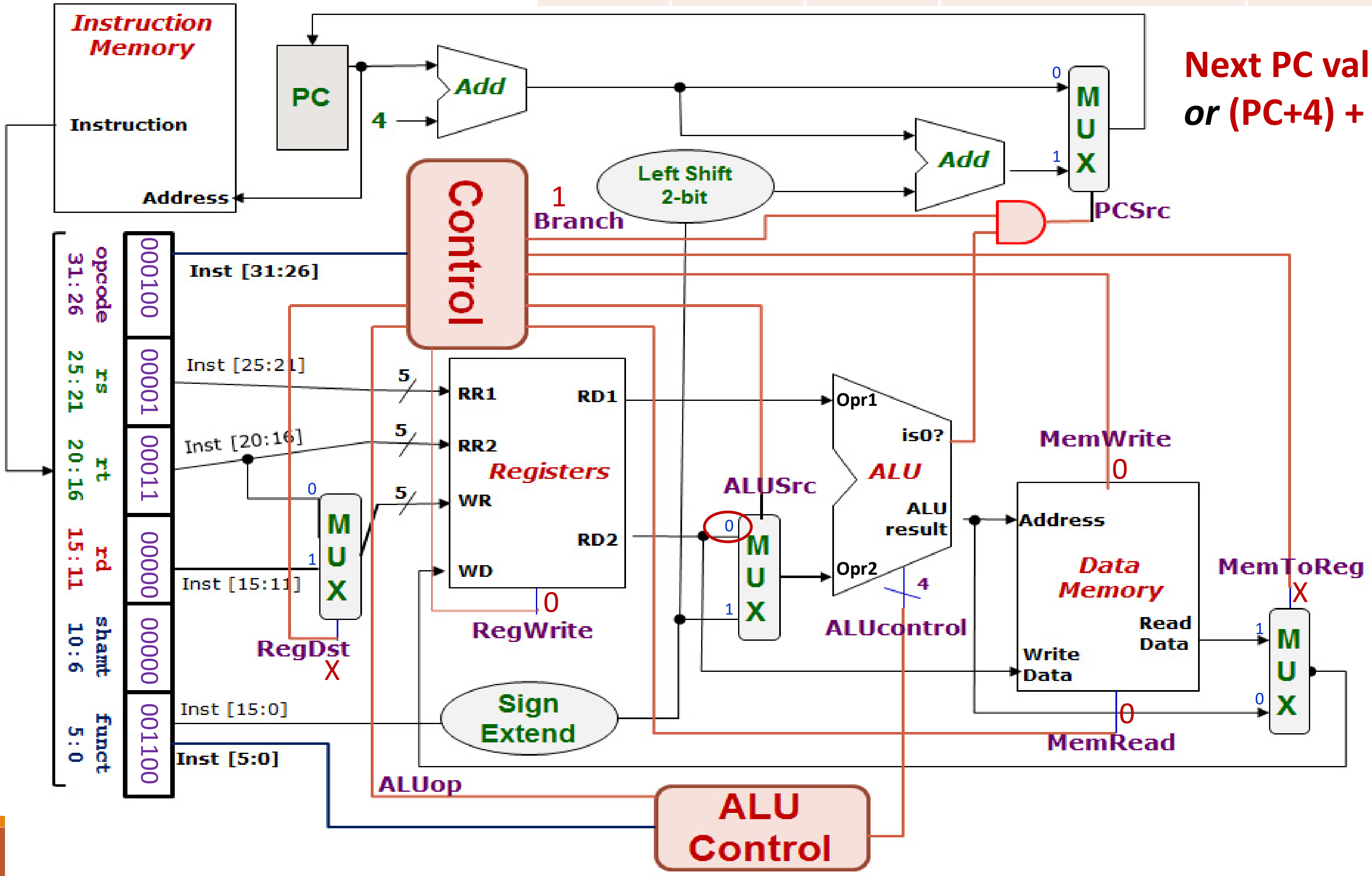
000100 00001 00011 0000000000001100

Q1(a)

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$1	$3	$3 or $0	[$1]-[$3] or MEM([$1]-[$3])	[$1]	[$3]	[$1] – [$3]	[$3]

Next PC value = PC + 4
***or* (PC+4) + (12×4)**

RegDest	X
RegWrite	0
ALUSrc	0
MemRead	0
MemWrite	0
MemToReg	X
Branch	1
ALUop	01
ALUcontrol	0110



`sub $25, $20, $5`

R[rd] = R[rs] – R[rt]

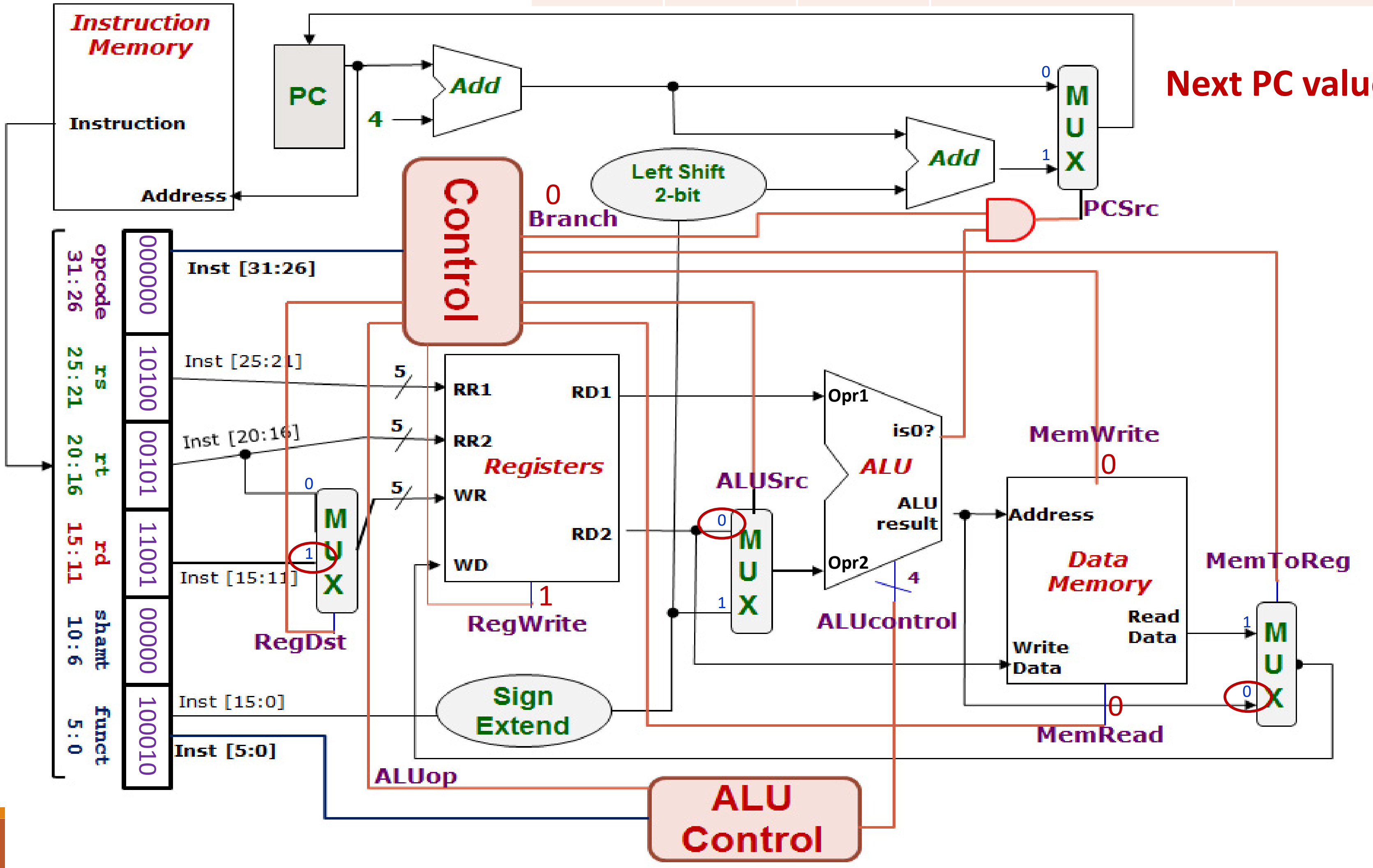
`000000 10100 00101 11001 00000 100010`

Q1(c)

Registers File				ALU		Data Memory	
RR1	RR2	WR	WD	Opr1	Opr2	Address	Write Data
$20	$5	$25	[$20] – [$5]	[$20]	[$5]	[$20] – [$5]	[$5]

Next PC value = PC + 4

RegDest	1
RegWrite	1
ALUSrc	0
MemRead	0
MemWrite	0
MemToReg	0
Branch	0
ALUop	10
ALUcontrol	0110



SUB instruction

Inst-Mem (400) → Reg.File (200) → MUX (ALUSrc) (30) → ALU (120) → MUX (MToR) (30) → Reg.File (200)

↘ Control (100)

Not critical path

Q2(a)

Inst-Mem
400ps

Adder
100ps

MUX
30ps

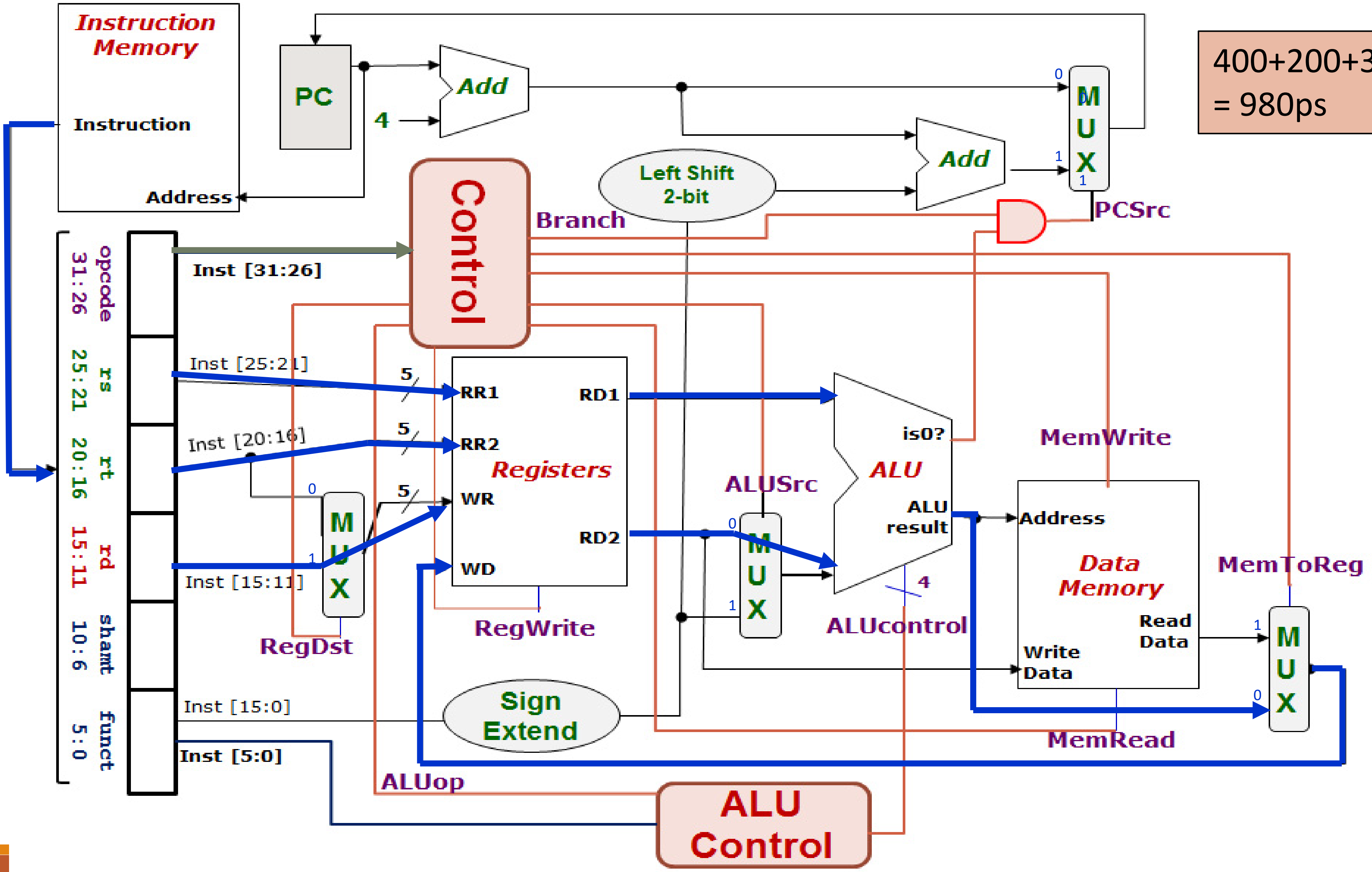
ALU
120ps

Reg-File
200ps

Data-Mem
350ps

Control/ALU Control
100ps

Lshft/signext/AND
20ps



400+200+30+120+30+200
= 980ps

LW instruction

$$\begin{tikzcd} \text{Inst-Mem (400)} \arrow[r] \arrow[dr] & \text{Reg.File (200)} \arrow[r] & \text{ALU (120)} \arrow[r] & \text{DataMem (350)} \arrow[r] & \text{MUX (MToR) (30)} \arrow[r] & \text{Reg.File (200)} \\ & \text{Control (100)} & & & & \end{tikzcd}$$

Not critical path

Q2(b)

Inst-Mem
400ps

Adder
100ps

MUX
30ps

ALU
120ps

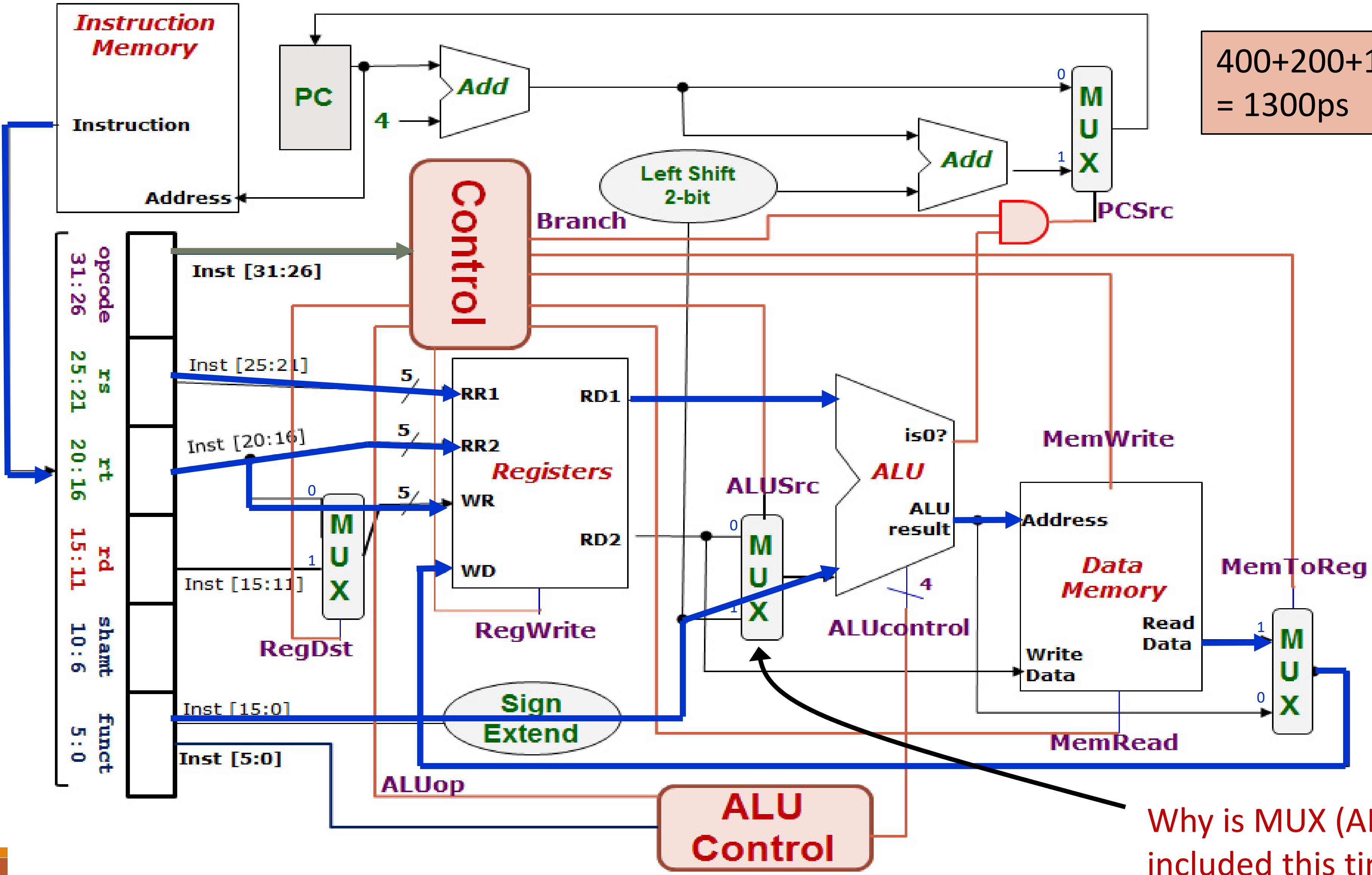
Reg-File
200ps

Data-Mem
350ps

Control/ALU
Control
100ps

Lshft/signext
/AND
20ps

400+200+120+350+30+200
= 1300ps



Why is MUX (ALUSrc) not included this time?

Prelude: "returning" multiple things in C

- Use pointers in argument to "return" multiple things

```
void func(int a, int b, int *ret1, int *ret2) {
    int y = a - b;
    *ret1 = a + b;
    *ret2 = y > 0 ? y : -1 * y;
}
```

Questions 3, 4, 5, 6, 7

- What's going on:
  - Imagine you're writing an emulator: you will need to simulate what the hardware does in code
- What this means:
  - Find the 'code' (in C for this module) that simulates what each of the components do
  - For example, we have the Instruction Memory:
    - "Text segment" from 0x0040 0000 to 0x1000 0000, so we have 0x0FC0 0000 bytes of Instr. Mem.
    - Represent that by `uint32_t instr_mem[264241152]`

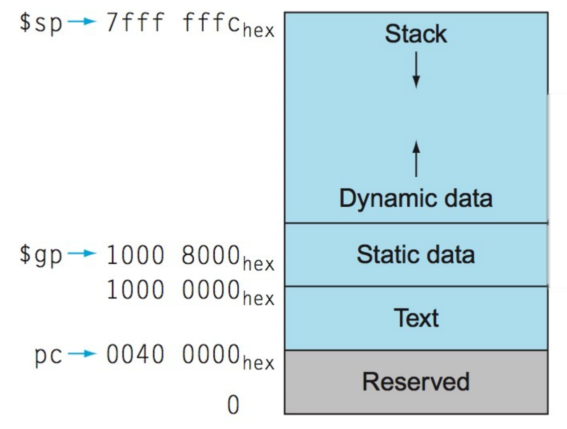
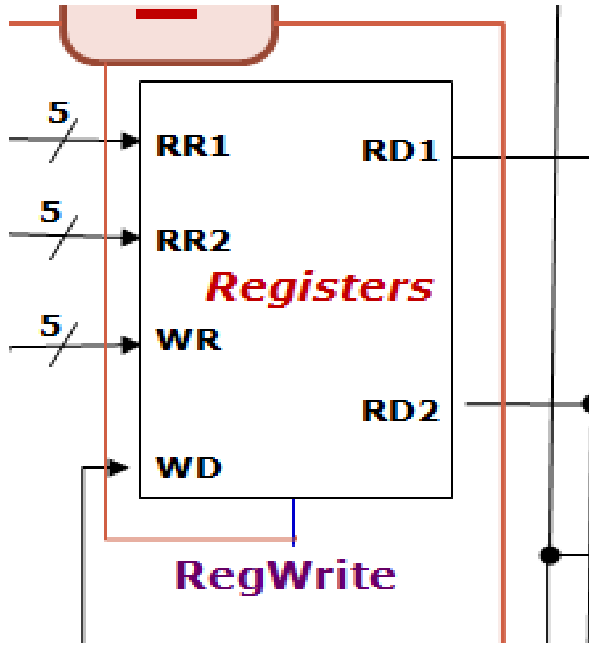


FIGURE 2.13 The MIPS memory allocation for program and data. These addresses are only a software convention, and not part of the MIPS architecture. The stack pointer is initialized to `7fff fffc`$_{hex}$ and grows down toward the data segment. At the other end, the program code ("text") starts at `0040 0000`$_{hex}$. The static data starts at `1000 0000`$_{hex}$. Dynamic data, allocated by `malloc` in C and by `new` in Java, is next. It grows up toward the stack in an area called the heap. The global pointer, `$gp`, is set to an address to make it easy to access data. It is initialized to `1000 8000`$_{hex}$ so that it can access from `1000 0000`$_{hex}$ to `1000 ffff`$_{hex}$ using the positive and negative 16-bit offsets from `$gp`. This information is also found in Column 4 of the MIPS Reference Data Card at the front of this book.

Q3a) Register file

Now we're simulating the register file.

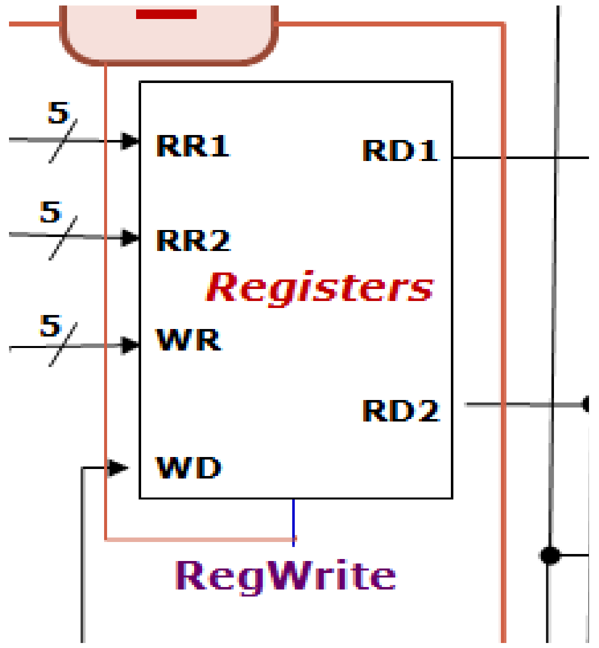
1. What is the data in the register file?
2. What stuff do we do with the register file?



Q3a) Register file

Now we're simulating the register file.

1. What is the data in the register file?
   - 32 registers, each with 32-bit field
2. What stuff do we do with the register file?
   - We get RR1/RR2/WR/WD/RegWrite
   - We pass RD1 and RD2 out
     - Remember how to "return" multiple things?
   - We might need to modify the register file
     - This happens when RegWrite is 1



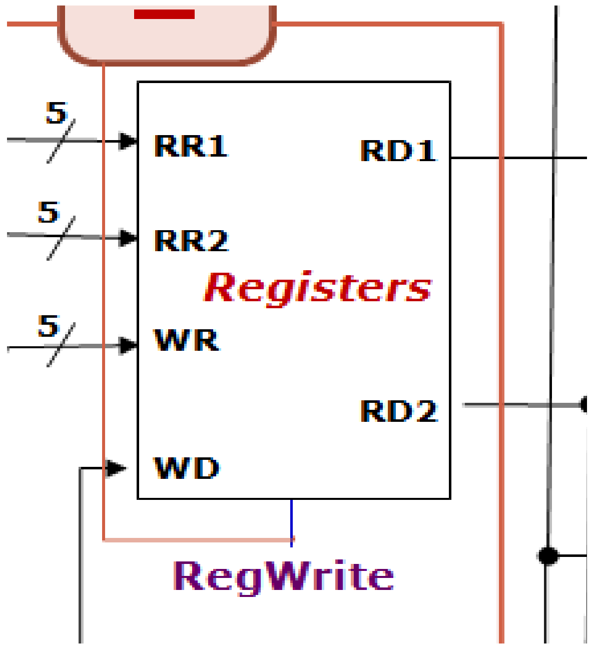
Q3a) Register file

Now we're simulating the register file.

1. What is the data in the register file?
    - 32 registers, each with 32-bit field

How to simulate this:

```
int32_t rf[32];
```



Q3a) Register file

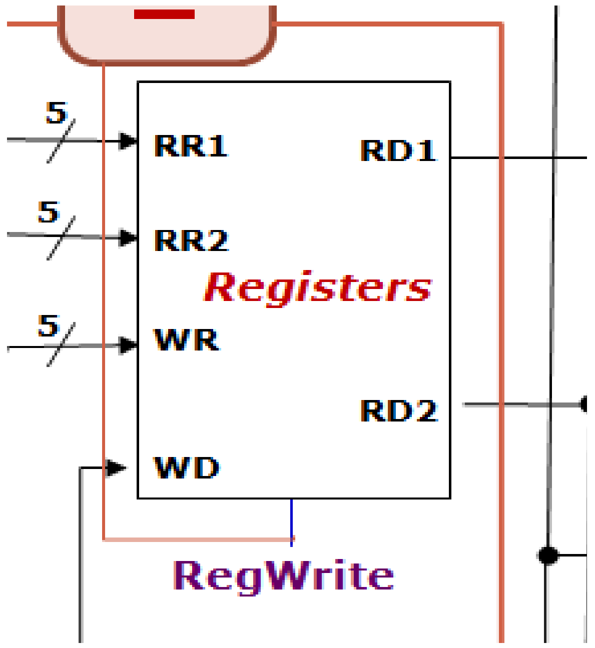
Now we're simulating the register file.

2. What stuff do we do with the register file?
   - We get RR1/RR2/WR/WD/RegWrite

     => simulated as arguments to function
   - We pass RD1 and RD2 out

     => simulated as "return" values
   - We might need to modify the register file

     => the body of the function



Q3a

```
int32_t rf[32];

void RegFile(uint5_t RR1,
             uint5_t RR2,
             uint5_t WR,
             int32_t WD,
             int32_t *RD1,
             int32_t *RD2,
             bool RegWrite) {

  // Because we need to send out multiple outputs,
  // we will use passing by pointers.
  *RD1 = (RR1 > 0) ? rf[RR1] : 0;
  *RD2 = (RR2 > 0) ? rf[RR2] : 0;

  if (RegWrite && WR)
    rf[WR] = WD;
}
```

Just an array of integers to store the registers content. The index shall be the name (i.e., register number) of the register

Since we need to return two outputs, we will need to pass them back via pointers

Make sure that register 0 returns a 0

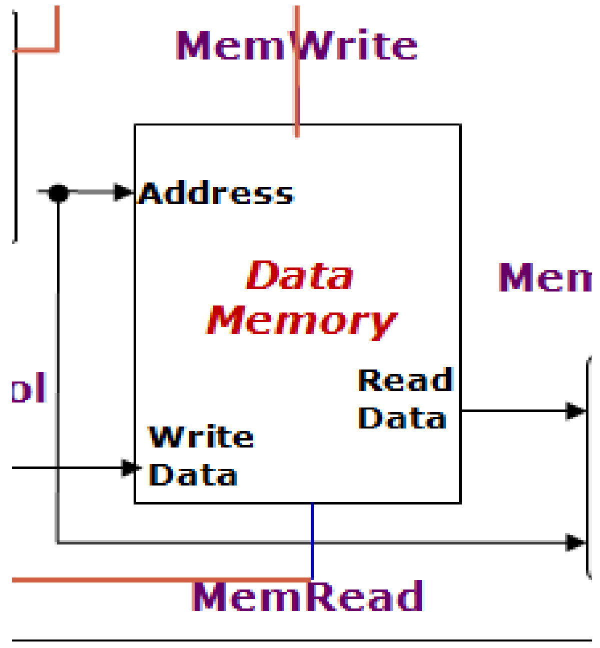
Make sure that we never write to register 0. Actually, harmless coz you can never read the content back. Still, just for completeness

We can use C's conditional statement or if-then-else. Both are perfectly fine

Q3b) Data Memory

Now we're simulating the data memory.

1. What is the data memory?
   - The simplification is always "a large array"
2. What stuff do we do with data memory?
   - Input: Address, Write Data
   - Input: MemWrite, MemRead
   - Output: ReadData



Q3b

```
int32_t data_memory[1073741824];

int32_t AccessDataMemory(uint32_t address,
                         int32_t WrData,
                         bool MemRead,
                         bool MemWrite)
{
     // We can do a sanity check here.
     // You can at most do one memory operation.
     // Will assume that "error" raises hell.
     if (MemRead && MemWrite) {
        error("Cannot do both read and write at the same time.")
     }

     if (MemRead) {
        return data_memory[address];
     }

     if (MemWrite) {
        data_memory[address] = WrData;
        return 0;
     }
 }
```

Ridiculous number but without introducing OS, we will just live with this for now

Simple error checking

Straightforward

Q4

Macro processing is kind of automated text processing of your source code before it is passed to the compiler. It is useful especially for shortening repetitive coding. Not just save space but also so that you only need to fix one piece of code instead of fixing all the repeated copies. In the repetition process, it can also allow the programmer to
e parts of the repetition on a case by case basis

Macro "MUX" defined with "N" as a parameter

he token
with the
er N and
then with "_t"
form a new to

ken
the
to

MU

In macro, one line, one statement. If what you want to write spans multiple lines, continuation of must be made explicit by "\"

```
#define MUX(N) \
        uint##N##_t mux##N##(uint##N##_t in0, \
                             uint##N##_t in1, \
                             bool ctrl)       \
        {                                     \
            if (ctrl) return in1;             \
            else return in0;                  \
        }
```

```
uint8_t mux8(uint8_t in0,
             uint8_t in1, bool ctrl)
{
    if (ctrl) return in1;
    else return in0;
};
```

This will be instantiated at where "MUX(8);" occurs in the source file, before passing it to the C compiler

The semicolon here is due to "MUX(8);" written with a semicolon. It is copied verbatim. It makes "MUX(8);" look like a valid C statement and when expanded the semicolon is harmless as the result is still acceptable in C

Q5

(a) _RegDst

```
if (!opcode) *_RegDst = 1;
else         *_RegDst = 0;
```

(b) _ALUSrc

```
if (!opcode || opcode == 4) *_ALUSrc = 0;
else                        *_ALUSrc = 1;
```

(c) _MEMRead

```
if (opcode == 0x23) *_MemRead = 1;
else                *_MemRead = 0;
```

(d) _ALUOp

```
switch (opcode) {
   case 0:    *_ALUOp = 2;
   case 0x23:              // lw
   case 0x2b: *_ALUOp = 0; // sw
   case 0x4:  *_ALUOp = 1; // beq
}
```

Q6

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

```
uint4_t ALUControl(uint2_t _ALUOp, uint6_t _funct) {
  if (_ALUOp == 2) { // R-type; need to decode funct
    switch (_funct) {
       case 0x20: return 2;     // add
       case 0x22: return 6;     // sub
       case 0x24: return 0;     // and
       case 0x25: return 1;     // or
       case 0x27: return 0xC;   // nor
       case 0x2a: return 7;     // slt
       default: // raise an error
    }
  }
  else { // non R-type
    if (_ALUOp == 0)
       return 2; // Ask ALU to add
    if (_ALUOp == 1)
       return 6; // Ask ALU to subtract
  }
}
```

Q7

ALUcontrol	Function
0000	AND
0001	OR
0010	add
0110	subtract
0111	slt
1100	NOR

```
int32_t ALU(int32_t in0, int32_t in1, uint4_t ALUcontrol, bool *ALUiszero)
{
    int32_t result;

    switch (ALUcontrol) {
       case 0:
          result = in0 & in1;
          break;
       case 1:
          result = in0 | in1;
          break;
       case 2:
          result = in0 + in1;
          break;
       case 6:
          result = in0 - in1;
          break;
       case 7:
          result = (int32_t)(in0 < in1);
          break;
       case 12:
          result = ~(in0 | in1);
          break;
    }

    *ALUiszero = (result == 0);
    return(result);
}
```

in1 may be rt or immed. But this is settled prior to invoking the ALU

Our input parameters are signed. So we are doing signed comparison here

ALUiszero passed by pointer and is always set based on the result

We use the bitwise complement (NOT) of C. Cannot use logical NOT “!”

End of Tutorial 5

- Slides uploaded on github.com/theodoreleebrant/TA-2425S1
- Email: theo@comp.nus.edu.sg
- Anonymous feedback: bit.ly/feedback-theodore (or scan on the right)

(Also reminder for me to take attendance)