

CSCI 2500 — Computer Organization
Group Project (document version 1.0) — Due 30 April 2023
MIPS Processor in C

- This project is due by the Midnight EST on the above date via a Submittity gradeable.
- This project is a *group assignment*. You can have groups of up to four people.
- Start the project early. You can ask questions during office hours, in the Submittity forum, and during your lab session.

1 Project Overview

For the group project, we'll be taking Lab 5 and Homework 5 to the logical conclusion – you'll be implementing a full gate-level circuit representing the datapath for a reduced, but still Turing complete, MIPS ISA. While we have already developed our ALU, the rest of the datapath and control needs to be implemented. For most of the instructions and datapath, implementation details are provided in depth throughout the Chapter 4 slides as well as the book. However, another aspect of the project will be to expand on the provided datapath by developing your own implementations for a few other instructions. See the template file (`project.c`) for more details.

2 Supported Instructions

Our ISA will include the following instructions:

Instruction	Type	Description	Input format	Operation
lw	I-type	Load Word	lw reg1 reg2 offset	reg1 = M[reg2+offset]
sw	I-type	Store Word	sw reg1 reg2 offset	M[reg2+offset] = reg1
beq	I-type	Branch on equal	beq reg1 reg2 offset	if (reg1 == reg2) PC += offset
addi	I-type	Add immediate	addi reg1 reg2 constant	reg1 = reg2 + constant
and	R-type	Logical AND	and reg1 reg2 reg3	reg1 = reg2 & reg3
or	R-type	Logical OR	or reg1 reg2 reg3	reg1 = reg2 reg3
add	R-type	Integer addition	add reg1 reg2 reg3	reg1 = reg2 + reg3
sub	R-type	Integer subtraction	sub reg1 reg2 reg3	reg1 = reg2 - reg3
slt	R-type	Set less than	slt reg1 reg2 reg3	reg1 = (reg2 < reg3 ? 1 : 0)
j	J-type	Jump	j address	PC = address
jal	J-type	Jump and link	jal address	RA = PC, PC = address
jr	R-type	Jump register	jr reg1	PC = reg1

The “input format” given above refers to the format of the assembly instructions you'll parse, convert to machine code, and then process through your circuit. **Note that for lw and sw the format is slightly different than what we've seen in the past.** This adjustment was made to simplify input processing. Also note that you'll need to map each register label (e.g., `t0`) to its 5-bit binary representation.

2.1 Instruction Conversion

Your first step is to parse the input instructions into their 32-bit binary machine code representation. See the MIPS reference card on Submittity for the instruction format fields, the op codes, function codes, and register mappings (rs, rt, rd) for each instruction. For Lab 6, you'll be implementing this for a subset of instruction types and only the t0 and s0 registers. For this project, you'll need to consider all of the following the registers:

Register Name	Register #	Use
zero	0	Constant value 0
v0	2	Return value register 0
a0	4	Argument register 0
t0	8	Temporary register 0
t1	9	Temporary register 1
s0	16	Saved register 0
s1	17	Saved register 1
sp	29	Stack pointer
ra	31	Return address

3 Memory

For simplicity, we'll have 3 separate 32×32 -bit 2D arrays representing storage for the register file, instruction memory, and data memory. You won't need to construct D-flip-flops or similar circuits to store program state. For additional simplicity, memory is to be addressed on the word (**not byte**) boundary. What this means is:

- For updating your PC, you'll only need to add +1 instead of +4.
- For jump and branch addresses, you won't need to shift left by 2.
- Any jump value in an instruction will be the actual address to jump to.

To access and read/write a specific memory address, you can use the 5-to-32 decoder to be implemented in Lab 6 along with read/write control bits. The different memories are implemented in 4 separate functions, including `InstructionMemory`, `ReadRegister`, `WriteRegister`, and `DataMemory`.

4 Control

There will be several sets of control bits that you'll need to determine for each instruction. These include the 4-bit ALU Control bits input to the ALU, the 2-bit ALUOp that is input into ALU Control circuit, as well as 9 other control bits. You'll primarily be calculating these values in the functions `Control` and `ALU_Control`. The control bits are all specified below.

Control Bit	Size	Use
ALUControl	4	Output from ALU Control circuit to ALU
RegDst	1	TRUE if R-type instruction
Jump	1	TRUE if jump instruction
Branch	1	TRUE if branch instruction
MemRead	1	TRUE if reading from memory
MemToReg	1	TRUE if writing to register from memory
ALUOp	2	Output from Control circuit to ALU Control circuit
MemWrite	1	TRUE if writing to memory
ALUSrc	1	TRUE if second input to ALU is immediate value
RegWrite	1	TRUE if register is being written
Zero	1	TRUE if ALU result is Zero

See the following tables for guidance on how to set these values for each instruction:

- ALUControl and ALUOp: Chapter-4a, slide 33; Figure 4.13 from book
- Others: Chapter-4a, slides 38-44; Figure 4.18, Figure 4.22 from book

However, **be careful** when implementing any SOP circuits using *only* the values in the provided tables. These tables do not include a few instructions that we're considering (j, jal, jr, addi). In addition, be aware that you'll likely to need to implement your own additional control lines for these specific instructions.

See below for Figure 4.13, Figure 4.18, and Figure 4.22 as copied from the book (5th Edition).

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	0010
X	1	X	X	X	X	X	X	0110
1	X	X	X	0	0	0	0	0010
1	X	X	X	0	0	1	0	0110
1	X	X	X	0	1	0	0	0000
1	X	X	X	0	1	0	1	0001
1	X	X	X	1	0	1	0	0111

FIGURE 4.13 The truth table for the 4 ALU control bits (called Operation). The inputs are the ALUOp and function code field. Only the entries for which the ALU control is asserted are shown. Some don't-care entries have been added. For example, the ALUOp does not use the encoding 11, so the truth table can contain entries 1X and X1, rather than 10 and 01. Note that when the function field is used, the first 2 bits (F5 and F4) of these instructions are always 10, so they are don't-care terms and are replaced with XX in the truth table.

Instruction	RegDst	ALUSrc	MemtoReg	RegWrite	MemRead	MemWrite	Branch	ALUOp1	ALUOp0
R-format	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

FIGURE 4.18 The setting of the control lines is completely determined by the opcode fields of the instruction. The first row of the table corresponds to the R-format instructions (`add`, `sub`, `AND`, `OR`, and `sllt`). For all these instructions, the source register fields are `rs` and `rt`, and the destination register field is `rd`; this defines how the signals `ALUSrc` and `RegDst` are set. Furthermore, an R-type instruction writes a register (`RegWrite` = 1), but neither reads nor writes data memory. When the Branch control signal is 0, the PC is unconditionally replaced with `PC + 4`; otherwise, the PC is replaced by the branch target if the Zero output of the ALU is also high. The `ALUOp` field for R-type instructions is set to 10 to indicate that the ALU control should be generated from the funct field. The second and third rows of this table give the control signal settings for `lw` and `sw`. These `ALUSrc` and `ALUOp` fields are set to perform the address calculation. The `MemRead` and `MemWrite` are set to perform the memory access. Finally, `RegDst` and `RegWrite` are set for a load to cause the result to be stored into the `rt` register. The branch instruction is similar to an R-format operation, since it sends the `rs` and `rt` registers to the ALU. The `ALUOp` field for branch is set for a subtract (`ALU control` = 01), which is used to test for equality. Notice that the `MemtoReg` field is irrelevant when the `RegWrite` signal is 0: since the register is not being written, the value of the data on the register data write port is not used. Thus, the entry `MemtoReg` in the last two rows of the table is replaced with X for don't care. Don't cares can also be added to `RegDst` when `RegWrite` is 0. This type of don't care must be added by the designer, since it depends on knowledge of how the datapath works.

Input or output	Signal name	R-format	lw	sw	beq
Inputs	Op5	0	1	1	0
	Op4	0	0	0	0
	Op3	0	0	1	0
	Op2	0	0	0	1
	Op1	0	1	1	0
	Op0	0	1	1	0
Outputs	RegDst	1	0	X	X
	ALUSrc	0	1	1	0
	MemtoReg	0	1	X	X
	RegWrite	1	1	0	0
	MemRead	0	1	0	0
	MemWrite	0	0	1	0
	Branch	0	0	0	1
	ALUOp1	1	0	0	0
	ALUOp0	0	0	0	1

FIGURE 4.22 The control function for the simple single-cycle implementation is completely specified by this truth table. The top half of the table gives the combinations of input signals that correspond to the four opcodes, one per column, that determine the control output settings. (Remember that `Op` [5:0] corresponds to bits 31:26 of the instruction, which is the `op` field.) The bottom portion of the table gives the outputs for each of the four opcodes. Thus, the output `RegWrite` is asserted for two different combinations of the inputs. If we consider only the four opcodes shown in this table, then we can simplify the truth table by using don't cares in the input portion. For example, we can detect an R-format instruction with the expression $\overline{\text{Op5}} \cdot \overline{\text{Op2}}$, since this is sufficient to distinguish the R-format instructions from `lw`, `sw`, and `beq`. We do not take advantage of this simplification, since the rest of the MIPS opcodes are used in a full implementation.

5 ALU, Adder, and Other Circuits

You'll need to implement additional circuitry for the ALU and 32-bit adders. You can mostly re-use your implementations from Lab 5 and Homework 5, with a couple caveats.

- There are now 4 ALU control input bits, instead of just 2
- We have an additional output for the control line Zero

You'll further need to implement a "sign-extend" circuit for instructions with a 16-bit immediate value field. You are also free to implement any additional "helper circuits", as you deem necessary. It will likely be useful to implement things like 3-input or 4-input AND/OR gates.

6 Datapath

The datapath describes the way that all of the above control lines and circuits are hooked up to each other. Recall that we're considering the entire processor as one big combinational circuit. Each sub-circuit is similarly a combinational circuit of some inputs mapping to some outputs. You'll need to implement the functionality of the datapath by feeding an output from one circuit to the input in another circuit in some specified order. A representative figure to base your implementation off of is given by slide 53 in Chapter-4a or in Figure 4.24 in the book. However, again be aware that this figure doesn't consider some of the additional instructions that we're including. As part of your project writeup, you'll need to show how you modified the datapath to account for these new instructions.

Figure 4.24 is provided below.

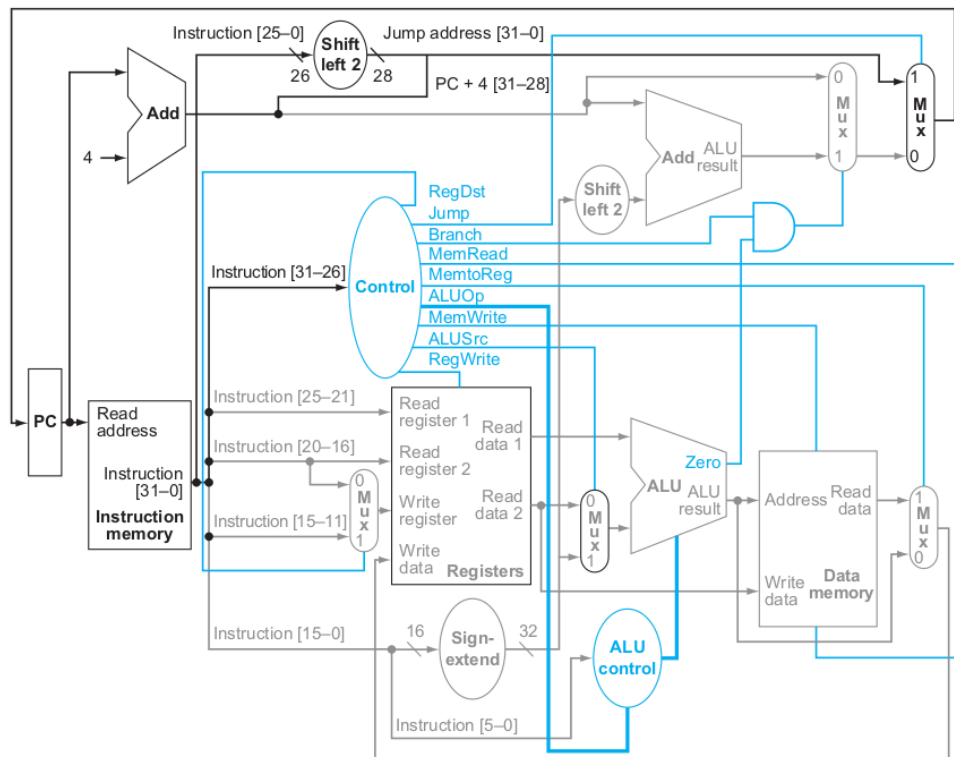


FIGURE 4.24 The simple control and datapath are extended to handle the jump instruction. An additional multiplexor (at the upper right) is used to choose between the jump target and either the branch target or the sequential instruction following this one. This multiplexor is controlled by the jump control signal. The jump target address is obtained by shifting the lower 26 bits of the jump instruction left 2 bits, effectively adding 00 as the low-order bits, and then concatenating the upper 4 bits of PC + 4 as the high-order bits, thus yielding a 32-bit address.

The datapath will primarily be implemented in the function `updateState()`. Although the processor we're implementing doesn't have explicit pipeline stages, it might still help you figure out your implementation by considering each pipeline stage at a time. E.g., You should process all of the circuits that would be in the DECODE stage of the pipeline before processing the circuits that would be in the EXECUTE stage.

7 Project Rules

The rules for your project implementation will be the same as with Homework 5. Basically, anything you code up for your processor should be representable as some actual combinational circuit. Generally: for loops over fixed iteration counts are OK, while all if-else statements are NOT. You are free to re-use any code from any group members' prior assignments. Group work should be as evenly distributed among members as is possible. Try and start as early as is reasonably possible.

8 Input and Output

The input format will be a list of assembly instructions. The output will be for each processed instruction: the PC, the binary instruction, the state of all 32 words of memory, and the state of all 32 registers. See below for example inputs and outputs:

```
bash$ cat arith.s
addi t0 zero 12
addi t1 zero 13
add s0 t0 t1
sub s1 t0 t1
and a0 t0 t1
or v0 t0 t1
bash$ ./a.out < arith.s
PC: 0
Instruction: 00100000000010000000000000001100
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 0 0 0 0 0 0 12 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0
PC: 1
Instruction: 00100000000010010000000000001101
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 0 0 0 0 0 0 12 13 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0
PC: 2
Instruction: 00000001000010011000000000100000
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 0 0 0 0 0 0 12 13 0 0 0 0 0 0 25 0 0 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0
PC: 3
Instruction: 00000001000010011000100000100010
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 0 0 0 0 0 0 12 13 0 0 0 0 0 0 25 -1 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0
```

```

PC: 4
Instruction: 0000000010000100100100000000100100
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 0 0 12 0 0 0 12 13 0 0 0 0 0 0 25 -1 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0
PC: 5
Instruction: 0000000010000100100010000000100101
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 13 0 12 0 0 0 12 13 0 0 0 0 0 0 25 -1 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0

```

```

bash$ cat load_store.s
addi t0 zero 25
add t1 t0 t0
addi sp sp -2
sw t0 sp 0
sw t1 sp 1
lw s0 sp 0
lw s1 sp 1
addi sp sp 2
bash$ ./a.out < load_store.s

```

```

PC: 0
Instruction: 0010000000000100000000000000011001
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 0 0 0 0 0 0 25 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0
PC: 1
Instruction: 0000000010000100001001000000100000
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 0 0 0 0 0 0 25 50 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 32 0 0
PC: 2
Instruction: 0010001110111101111111111111111110
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
Register: 0 0 0 0 0 0 0 0 25 50 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 30 0 0
PC: 3
Instruction: 1010111110101000000000000000000000
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 0
Register: 0 0 0 0 0 0 0 0 25 50 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 30 0 0
PC: 4
Instruction: 1010111110101001000000000000000001
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 50
Register: 0 0 0 0 0 0 0 0 25 50 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 30 0 0
PC: 5
Instruction: 1000111110110000000000000000000000
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 50
Register: 0 0 0 0 0 0 0 0 25 50 0 0 0 0 0 0 25 0 0 0 0 0 0 0 0 0 0 0 0 30 0 0
PC: 6
Instruction: 1000111110110001000000000000000001
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 50
Register: 0 0 0 0 0 0 0 0 25 50 0 0 0 0 0 0 25 50 0 0 0 0 0 0 0 0 0 0 30 0 0
PC: 7

```

```
Instruction: 00100011101111010000000000000010
Data: 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 25 50
Register: 0 0 0 0 0 0 0 0 25 50 0 0 0 0 0 0 25 50 0 0 0 0 0 0 0 0 0 0 32 0 0
```

9 Submission and Grading Criteria

Due to the complexity and duration of the project, there will be a few different checkpoints and deadlines that you'll need to be aware of. The final submission will consist of two parts – one being the autograded component and the other being the project writeup.

- April 5 - Lab 6: Instruction parsing and 32-address memory implementation
- April 12 - Finalize your groups in Submitty
- April 28 - Final submissions to Submitty for autograded portion
- April 30 - Final submissions to Submitty for project writeup

9.1 Grading Breakdown

The grading breakdown is as follows:

1. Autograding: 50%
 - Standard visible and hidden test cases
2. TA grading: 50%
 - Solution is representative of an implementable circuit
 - Project writeup
 - Fully explains design and implementation choices for each circuit
 - Includes modified datapath from Figure 4.24 for new instructions
 - Explains contributions of each group member