RISC-V Single Cycle Microarchitecture

Project By:

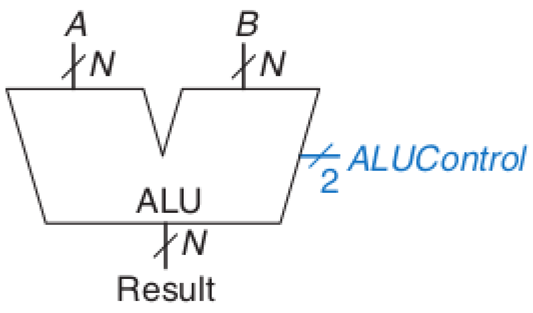
MERL-DSU

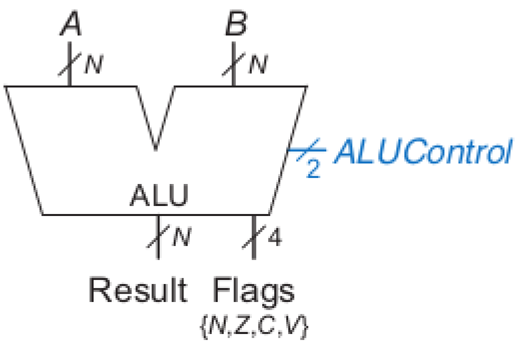
Table of Contents

1.1 Arithmetic Logic Unit (ALU)

1.1.1 Introduction

We will design an ALU that can perform a subset of the ALU operations of a full Processor ALU.





In this exercise, we will develop an ALU that will take two 2-inputs, A and B and is able to execute the following instructions:

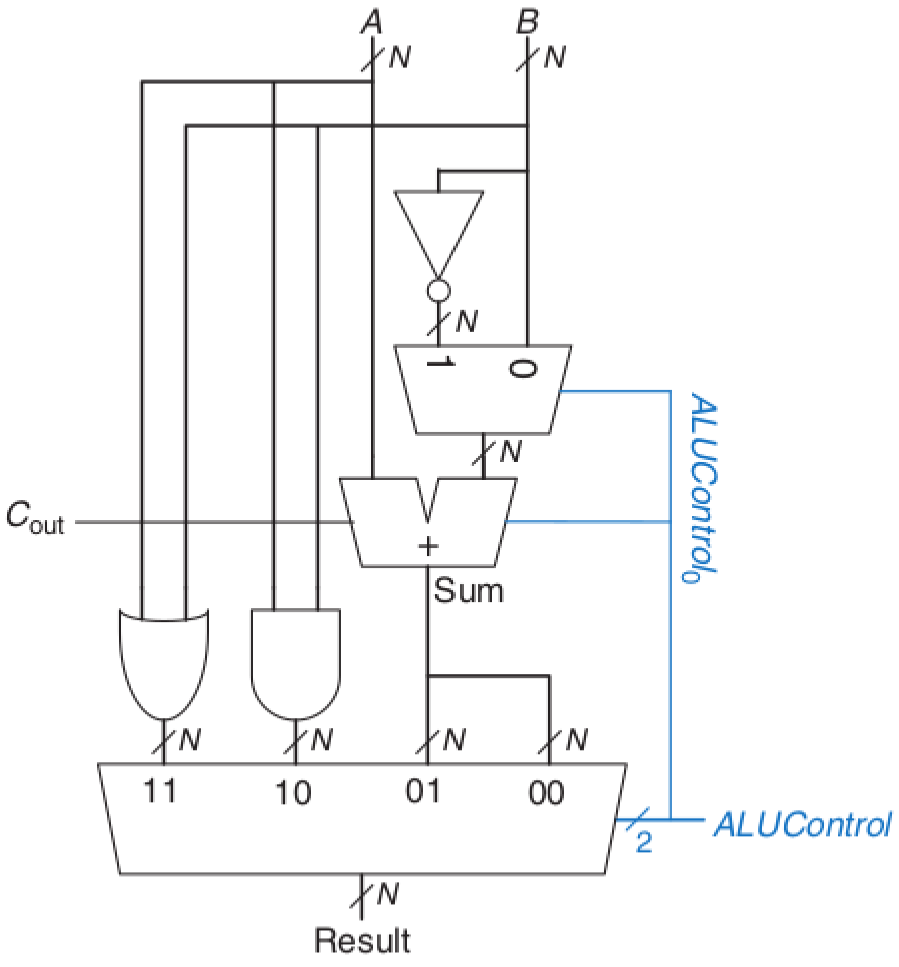
ALUControl	Instruction
000 (add)	`lw, sw`
001 (subtract)	`beq`
000 (add)	`add`
001 (subtract)	`sub`
101 (set less than)	`slt`
011 (or)	`or`
010 (and)	`and`

The ALU will generate a 32-bit output that we will call ‘Result’ and an additional 1-bit flag ‘Zero’ that will be set to ‘logic-1’ if all the bits of ‘Result’ are 0. The different operations will be selected by a 3-bit control signal called ‘ALUControl’ according to the following table.

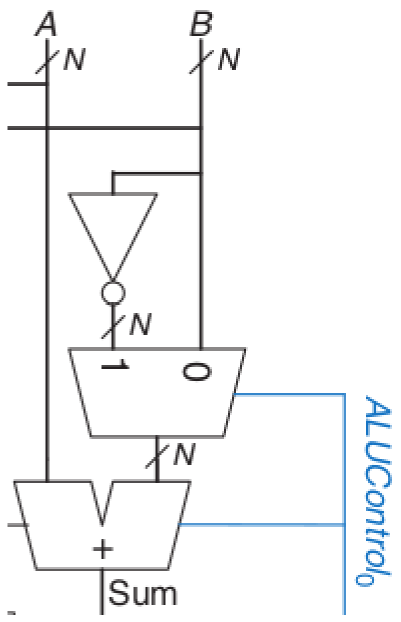
For example, when the ‘ALUControl’ input is ‘011’, the function Result = A or B should be calculated. It is easy to see that there are many values of ‘ALUControl’ for which no operation has been defined. It is not very important what the circuit does when ‘ALUControl’ has these values, since the ‘Result’ will simply be ignored in these cases. You can use this to your advantage to simplify the circuit. Right now, the described operations may look random, but once we learn more about the Instruction set architecture, these choices will make more sense.

1.1.2 Block diagram

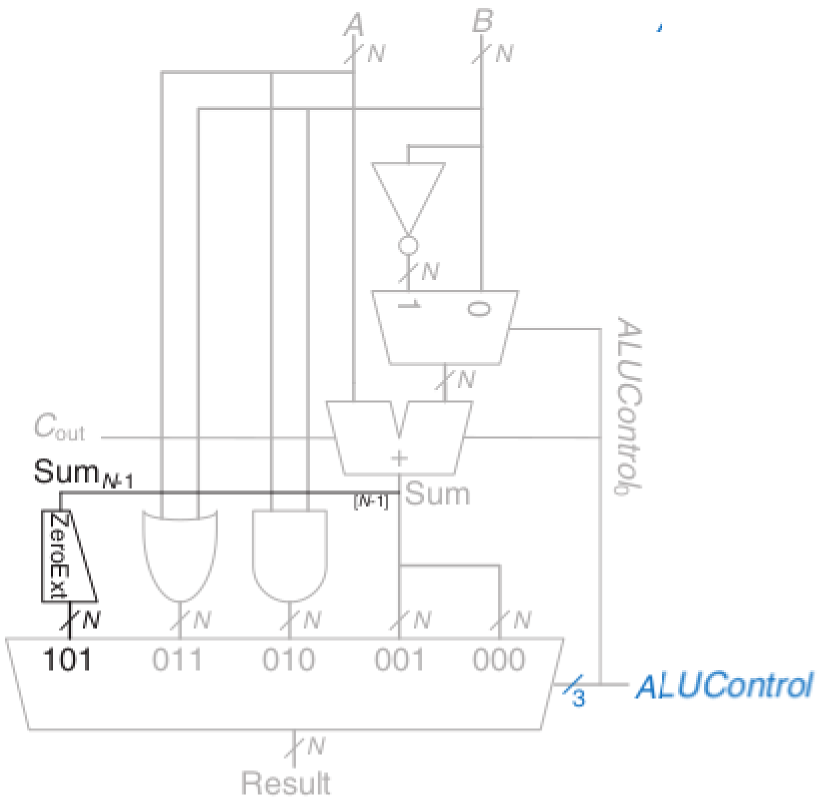
The first order of business should be drawing a block diagram. The following is one approach to analyzing what is needed and come up with a block diagram. You are free to follow this example or come up with your own ideas. It is just important that you think about how the circuit should be implemented. Let us first examine the different commands. You should see that we have two types of instructions. The three instructions **add, sub,** and **slt** are arithmetic operations, whereas the two remaining **and, or** are logical operations. Therefore, we have two separate groups of operations. Now let us look at the figure above and determine for which values of ALUControl we perform an operation from which group. It should be pretty clear that when ALUControl[1] is logic-1 we select a logic operation and when ALUControl[1] is logic-0 we have an arithmetic operation. This means that the output of either group can be selected by a 4-input multiplexer that is controlled by ALUControl[1:0]. Figure below shows this distribution.



Now we can take a look at the two groups individually. In the first group we realize that we have an addition (add) or a subtraction (sub, slt). We again could observe that ALUControl[0] is logic-0 for additions and logic-1 for subtractions.. Figure below shows this arrangement.

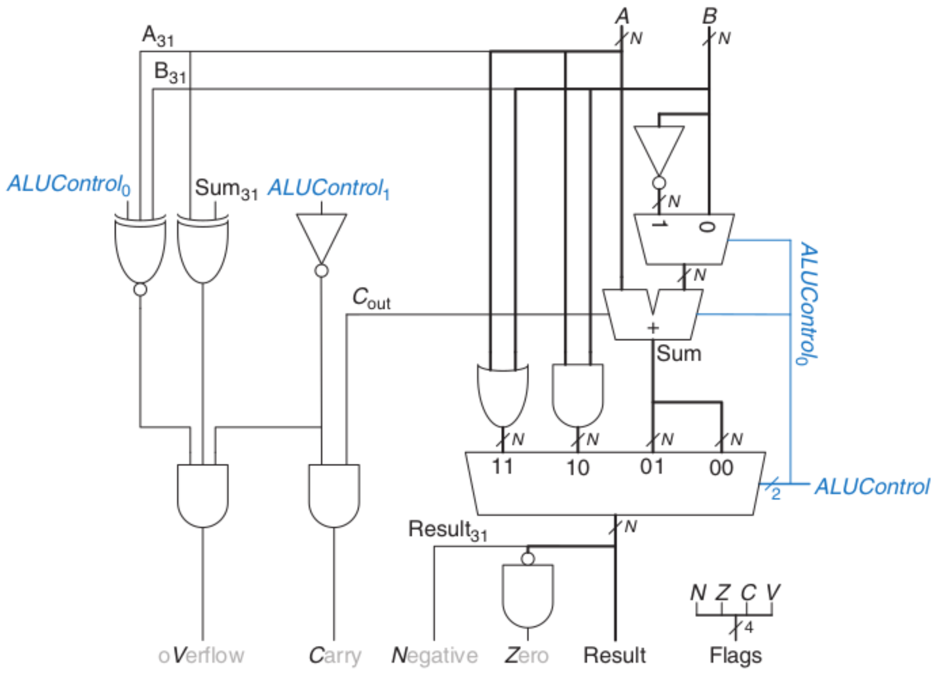


There is one more thing left, depending on the ALUControl[2] we could select whether we take only the most significant bit (logic-1, slt instruction), or we take the output as it is.



Finally, we also have to add a small circuit that will generate the zero output when the result equals all zeroes. This method of breaking a larger block successively into smaller pieces is called Divide and Conquer and is one of the most important tricks that allow hardware engineers to design very complex circuits.

Note that in the above example, there are many values of ALUControl where the circuit would perform 'strange' operations (100 for example). This is not important because the circuit specification that was given to us said that these inputs were not relevant (this is probably because it can be guaranteed that these inputs will not appear during normal operation).



Comparison	Signed	Unsigned
$=$	Z	Z
$\neq$	$\overline{Z}$	$\overline{Z}$
$<$	$N \oplus V$	$\overline{C}$
$\leq$	$Z + (N \oplus V)$	$Z + \overline{C}$
$>$	$\overline{Z} \bullet (\overline{N \oplus V})$	$\overline{Z} \bullet C$
$\geq$	$(\overline{N \oplus V})$	C

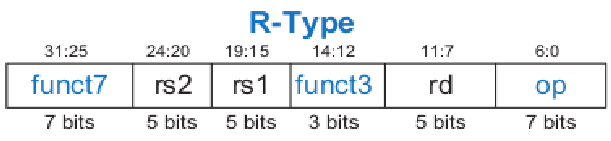
2.1 Control Unit

2.1.1 Introduction

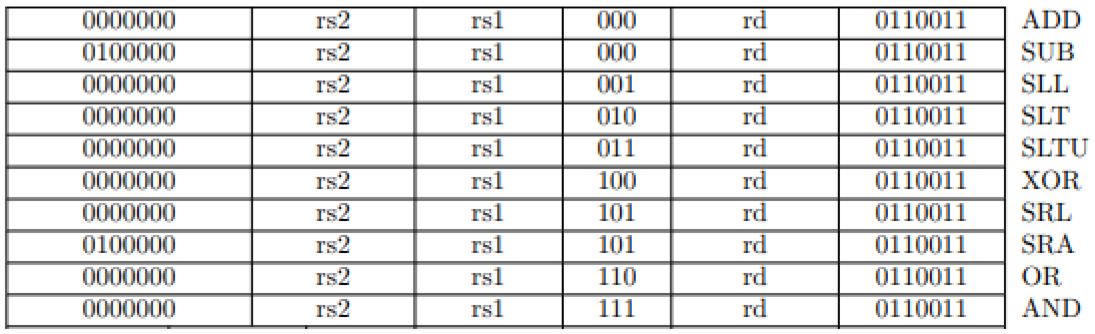
RISC-V uses 32-bit instructions. RISC-V consists of defining the following instruction formats: R-type, I-type, S-Type, B-Type, U-type, and J-type. R-type instructions operate on three registers. I-type, S-type and B-type instructions operate on two registers and a 12-bit immediate. U-type and J-type (jump) instructions operate on one 20-bit immediate.

2.1.1.1 R-Type Instructions

The name R-type is short for register-type. R-type instructions use three registers as operands: two as sources, and one as a destination. The figure below shows the R-type machine instruction format. The 32-bit instruction has six fields: op, rs1, rs2, rd, funct3, and funct7. Each field is five or seven bits, as indicated. The operation the instruction performs is encoded in the three fields highlighted in blue: **op** (also called opcode or operation code) and **funct3** and **funct7** (also called the function). All R-type instructions have an opcode of **33**. The specific R-type operation is determined by the function fields. The operands are encoded in the three fields: **rs1**, **rs2**, and **rd**. The first two registers, rs1, and rs2 are the source registers; rd is the destination register.



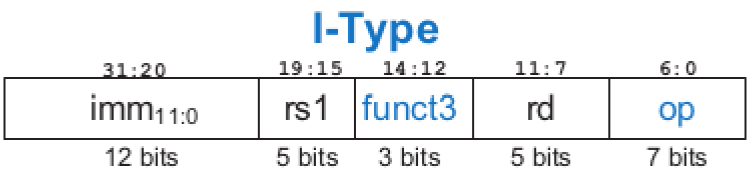
Instruction Format for some of the R-type instructions is shown below:



0000000	rs2	rs1	000	rd	0110011	ADD
0100000	rs2	rs1	000	rd	0110011	SUB
0000000	rs2	rs1	001	rd	0110011	SLL
0000000	rs2	rs1	010	rd	0110011	SLT
0000000	rs2	rs1	011	rd	0110011	SLTU
0000000	rs2	rs1	100	rd	0110011	XOR
0000000	rs2	rs1	101	rd	0110011	SRL
0100000	rs2	rs1	101	rd	0110011	SRA
0000000	rs2	rs1	110	rd	0110011	OR
0000000	rs2	rs1	111	rd	0110011	AND

2.1.1.2 I-Type Instructions

The name I-type is short for immediate-type. I-type instructions use two register operands and one immediate operand. The figure below shows the I-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rd, funct3, and imm. The first three fields, op, rs1, and rd, are like those of R-type instructions. The imm field holds the 12-bit immediate. The funct3 holds the operation to be performed. The operation is determined by the opcode and funct3, highlighted in blue. The operands are specified in the two fields rs1, and imm. rs1 and imm are always used as source operands. rd is used as a destination.

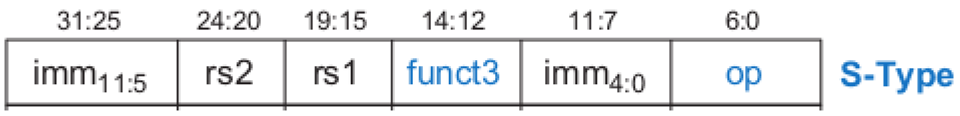


Instruction format for some of the I-type instructions are shown below:

imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI
0000000	shamt	rs1	001	rd	0010011	SLLI
0000000	shamt	rs1	101	rd	0010011	SRLI
0100000	shamt	rs1	101	rd	0010011	SRAI

2.1.1.3 S-Type Instructions

The name S-type is short for store-type. S-type instructions use two register operands and one immediate operand. Figure below shows the S-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rs2, funct3 and imm. The first three fields op, rs1, and rs2 are like those of R-type instructions. The imm fields hold the 12-bit immediate. The 12-bit immediate is split into two sets as shown below. The operation is determined by the opcode and funct3, highlighted in blue. The operands are specified in the three fields rs1, rs2 and imm.



Instruction format for some of the S-type instructions are shown below:

imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW

2.1.1.4 B-Type Instructions

The name B-type is short for branch-type. This format is used only with branch instructions. B-type instructions use two register operands and one immediate operand. Figure below shows the B-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rs2, funct3 and imm. The first three fields op, rs1, and rs2 are like those of R-type instructions. The imm fields hold the 12-bit immediate. The 12-bit immediate is split into two sets as shown below. The operation is determined by the opcode and funct3, highlighted in blue. The operands are specified in the three fields rs1, rs2 and imm.

$imm_{12,10:5}$	rs2	rs1	funct3	$imm_{4:1,11}$	op	B-Type
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	

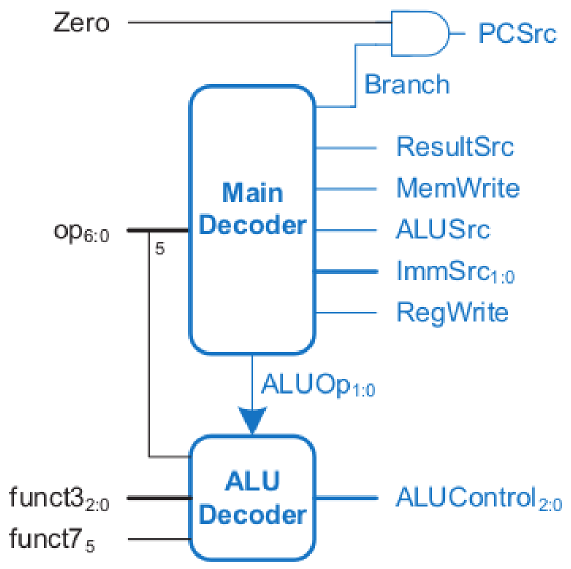
Instruction format for some of the B-type instructions are shown below:

imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011	BEQ
imm[12|10:5]	rs2	rs1	001	imm[4:1|11]	1100011	BNE
imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011	BLT
imm[12|10:5]	rs2	rs1	101	imm[4:1|11]	1100011	BGE
imm[12|10:5]	rs2	rs1	110	imm[4:1|11]	1100011	BLTU
imm[12|10:5]	rs2	rs1	111	imm[4:1|11]	1100011	BGEU

To interpret machine language, one must decipher the fields of each 32-bit instruction word. Different instructions use different formats, but all formats start with a 7-bit opcode field. Thus, the best place to begin is to look at the opcode.

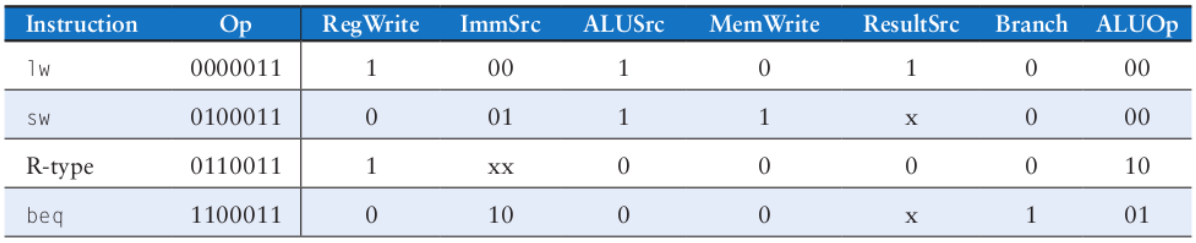
2.1.2 Block Diagram

The control unit computes the control signals based on the opcode and funct fields of the instruction, Instr[31:25], Instr[14:12] and Instr[6:0]. Most of the control information comes from the opcode, but for further operations function fields are used. Thus, we will simplify our design by factoring the control unit into two blocks of combinational logic, as shown in Figure below



2.1.2.1 MAIN Decoder

The table below is a truth table for the main decoder that summarizes the control signals as a function of the opcode. All R-type instructions use the same main decoder values; they differ only in the ALU decoder output. Recall that, for instructions that do not write to the register file (e.g., S-type and B-type), the ResultSrc control signals is don't care (X); the address and data to the register write port do not matter because RegWrite is not asserted. The logic for the decoder can be designed using your favorite techniques for combinational logic design.



Instruction	Op	RegWrite	ImmSrc	ALUSrc	MemWrite	ResultSrc	Branch	ALUOp
`lw`	0000011	1	00	1	0	1	0	00
`sw`	0100011	0	01	1	1	x	0	00
R-type	0110011	1	xx	0	0	0	0	10
`beq`	1100011	0	10	0	0	x	1	01

2.1.2.2 ALU Decoder

The main decoder computes most of the outputs from the opcode. It also determines a 2-bit ALUOp signal. The ALU decoder uses this ALUOp signal in conjunction with the funct field and opcode bit to compute ALUControl. The meaning of the ALUOp signal is given in Table below

ALUOp	Meaning
00	add
01	subtract
10	look at funct fields and opcode bit
11	N/A

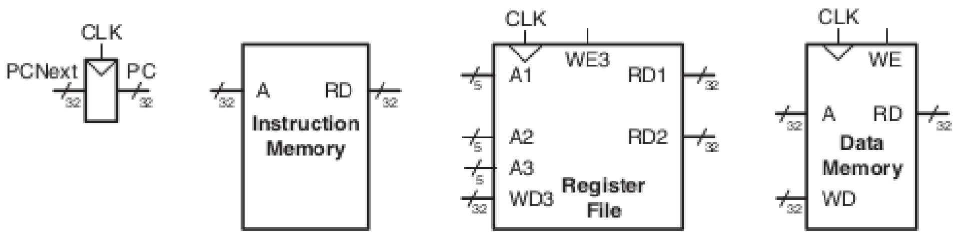
Table below is a truth table for the ALU decoder. The logic of ALUControl was covered in the above chapter. When ALUOp is 00 or 01, the ALU should add or subtract, respectively. When ALUOp is 10, the decoder examines the function fields and operand bit to determine the ALUControl. The control signals for each instruction were described as we built the datapath.

ALUOp	funct3	$\{op_5, funct7_5\}$	ALUControl	Instruction
00	x	x	000 (add)	`lw`, `sw`
01	x	x	001 (subtract)	`beq`
10	000	00, 01, 10	000 (add)	`add`
	000	11	001 (subtract)	`sub`
	010	x	101 (set less than)	`slt`
	110	x	011 (or)	`or`
	111	x	010 (and)	`and`

3.1 Designing Microarchitecture-I

3.1.1 State Elements

A good way to design a complex system is to start with hardware containing the state elements. These elements include the memories and the architectural state (the program counter and registers). Then, add blocks of combinational logic between the state elements to compute the new state based on the current state. The instruction is read from part of memory; load and store instructions then read or write data from another part of memory. Hence, it is often convenient to partition the overall memory into two smaller memories, one containing instructions and the other containing data. Figure below shows a block diagram with the four state elements: the program counter, register file, and instruction and data memories.



In above Figure heavy lines are used to indicate 32-bit data busses. Medium lines are used to indicate narrower busses, such as the 5-bit address busses on the register file. Narrow blue lines are used to indicate control signals, such as the register file write enable. We will use this convention throughout the chapter to avoid cluttering diagrams with bus widths. Also, state elements usually have a reset input to put them into a known state at start-up. Again, to save clutter, this reset is not shown.

The **program counter** is an ordinary 32-bit register. Its output, PC, points to the current instruction. Its input, PCNext, indicates the address of the next instruction.

The **instruction memory** has a single read port. It takes a 32-bit instruction address input, A, and reads the 32-bit data (i.e., instruction) from that address onto the read data output, RD.

The 32-element × 32-bit **register file** has two read ports and one write port. The read ports take 5-bit address inputs, A1 and A2, each specifying one of $2^5 = 32$ registers as source operands. They read the 32-bit register values onto read data outputs RD1 and RD2, respectively. The write port takes a 5-bit address input, A3; a 32-bit write data input, WD; a write enable input, WE3; and a clock. If the write enable is 1, the register file writes the data into the specified register on the rising edge of the clock.

The figure below shows the 32 registers available in RISC-V Microarchitecture.

Name	Register Number	Use
zero	x0	Constant value 0
ra	x1	Return address
sp	x2	Stack pointer
gp	x3	Global pointer
tp	x4	Thread pointer
t0–2	x5–7	Temporary registers
s0/fp	x8	Saved register / Frame pointer
s1	x9	Saved register
a0–1	x10–11	Function arguments / Return values
a2–7	x12–17	Function arguments
s2–11	x18–27	Saved registers
t3-6	x28–31	Temporary registers

The **data memory** has a single read/write port. If the write enable, WE, is 1, it writes data WD into address A on the rising edge of the clock. If the write enable is 0, it reads address A onto RD.

The instruction memory, register file, and data memory are all read combinationally. In other words, if the address changes, the new data appears at RD after some propagation delay; no clock is involved. They are written only on the rising edge of the clock. In this fashion, the state of the system is changed only at the clock edge. The address, data, and write enable must be set up sometime before the clock edge and must remain stable until a hold time after the clock edge. Because the state elements change their state only on the rising edge of the clock, they are synchronous sequential circuits.

The microprocessor is built of clocked state elements and combinational logic, so it too is a synchronous sequential circuit. Indeed, the processor can be viewed as a giant finite state machine, or as a collection of simpler interacting state machines.

4.1 Designing Microarchitecture-II

4.1.1 Introduction

Load word (lw) is an I-type instruction. The **LW instruction** loads data from the data memory through a specified address, with a possible offset, to the destination register. The name I-type is short for immediate-type. I-type instructions use two register operands and one immediate operand. Figure below shows the I-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rd, funct3 and imm. The first three fields, op, rs1, and rd, are like those of R-type instructions. The imm field holds the 12-bit immediate. The operation is determined by the opcode and funct3, highlighted in blue. The operands are specified in the two fields rs1, rd, and imm. rs1 and imm are always used as source operands. rd is used as a destination.

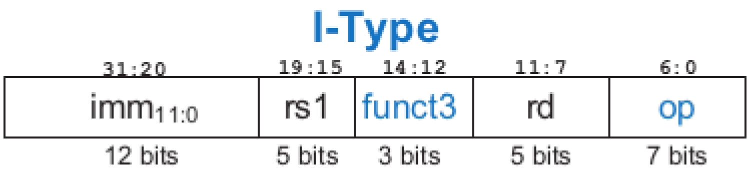
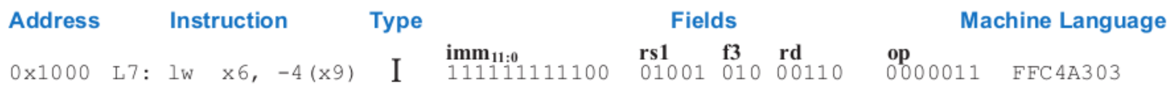


Figure below shows an example for LW instruction represented in assembly code and machine code.



Address	Instruction	Type	imm$_{11:0}$	rs1	f3	rd	op	Machine Language
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

I-type instructions have a 12-bit immediate field, but the immediates are used in 32-bit operations. For example, lw adds a 12-bit offset to a 32-bit base register. What should go in the upper half of the 32 bits?

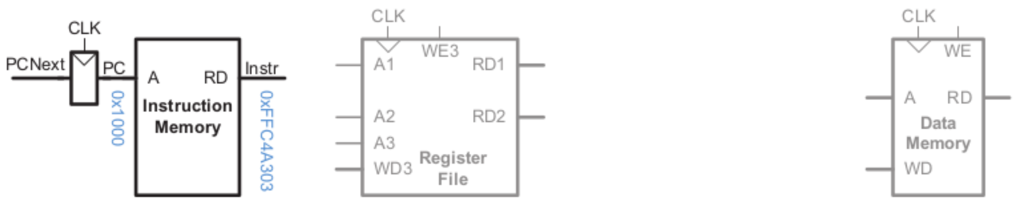
For positive immediates, the upper half should be all 0's, but for negative immediates, the upper half should be all 1's. This is called sign extension. An N-bit two's complement number is sign-extended to an M-bit number (M >N) by copying the sign bit (most significant bit) of the N-bit number into all of the upper bits of the M-bit number. Sign-extending a two's complement number does not change its value.

Example:
A = 0000 1010 0101
B = 1010 1100 1101
Zero Extension of A = 0000 0000 0000 0000 0000 0000 1010 0101
Zero Extension of B = **0000 0000 0000 0000 0000** 1010 1100 1101

A = 0101 0000 1101
B = 1011 1100 1111
Sign Extension of A = 0000 0000 0000 0000 0000 0101 0000 1101
Sign Extension of B = 1111 1111 1111 1111 1111 1011 1100 1111

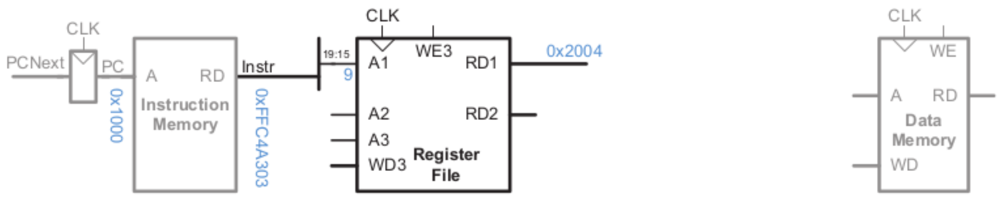
4.1.2 DataPath for Load Word Instruction

This section gradually develops the single-cycle datapath, adding one piece at a time to the state elements. The program counter (PC) register contains the address of the instruction to execute. The first step is to read the instruction from instruction memory. Figure below shows that the PC is simply connected to the address input of the instruction memory. The instruction memory reads out, or fetches, the 32-bit instruction, labeled Instr.



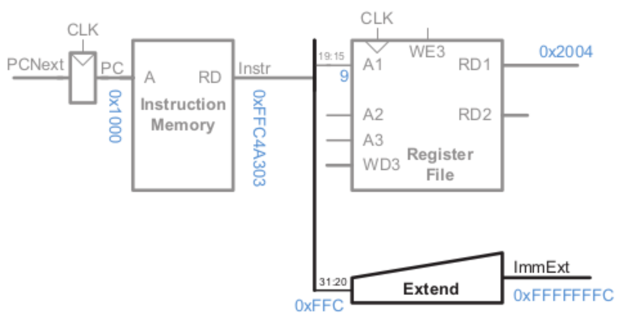
Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

For a load instruction, the next step is to read the source register containing the base address. This register is specified in the rs1 field of the instruction, Instr[19:15]. These bits of the instruction are connected to the address input of one of the register file read ports, A1, as shown in Figure below. The register file reads the register value onto RD1.



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

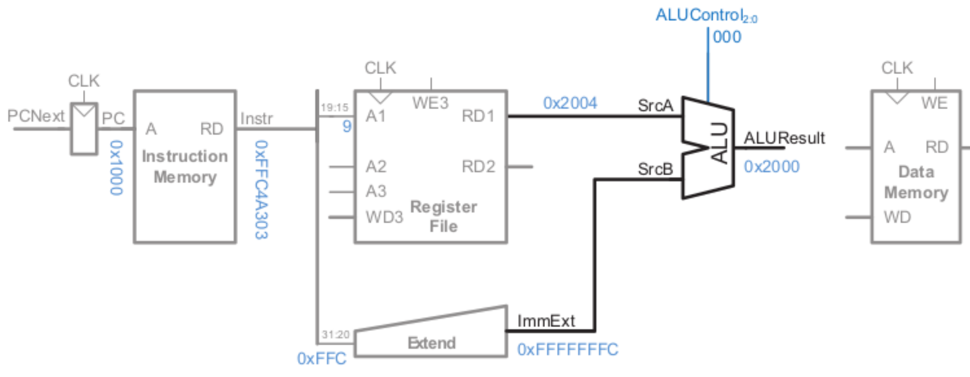
The load instruction also requires an offset. The offset is stored in the immediate field of the instruction, Instr[31:20]. Because the 12-bit immediate might be either positive or negative, it must be sign-extended to 32 bits, as shown in Figure below.



Address	Instruction	Type	Fields					Machine Language
			imm$_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

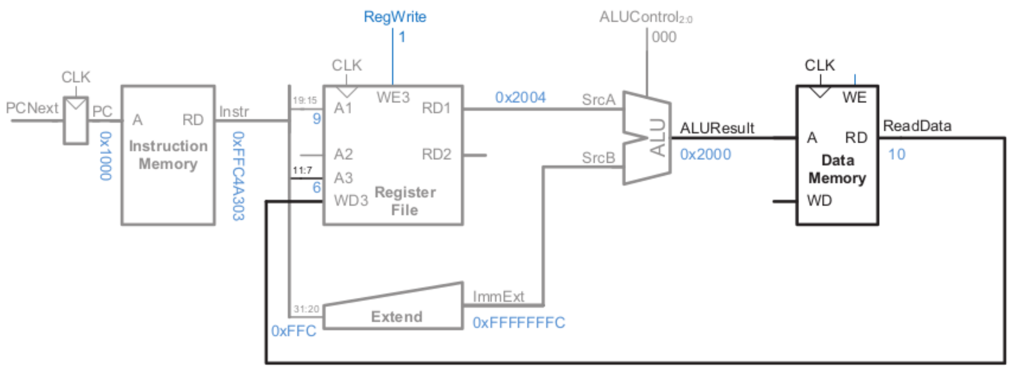
The 32-bit sign-extended value is called ImmExt. The sign extension simply copies the sign bit (most significant bit) of a short input into all of the upper bits of the longer output. Specifically, ImmExt[15:0] = Instr[31:20] and ImmExt[31:16] = Instr[31].

The processor must add the base address to the offset to find the address to read from memory. Figure below introduces an ALU to perform this addition. The ALU receives two operands, SrcA and SrcB. SrcA comes from the register file, and SrcB comes from the sign-extended immediate. The ALU can perform many operations. The 3-bit ALUControl signal specifies the operation. The ALU generates a 32-bit ALUResult and a Zero flag, which indicates whether ALUResult == 0. For a load instruction, the ALUControl signal should be set to 000 to add the base address and offset.



Address	Instruction	Type	Fields					Machine Language
			imm$_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

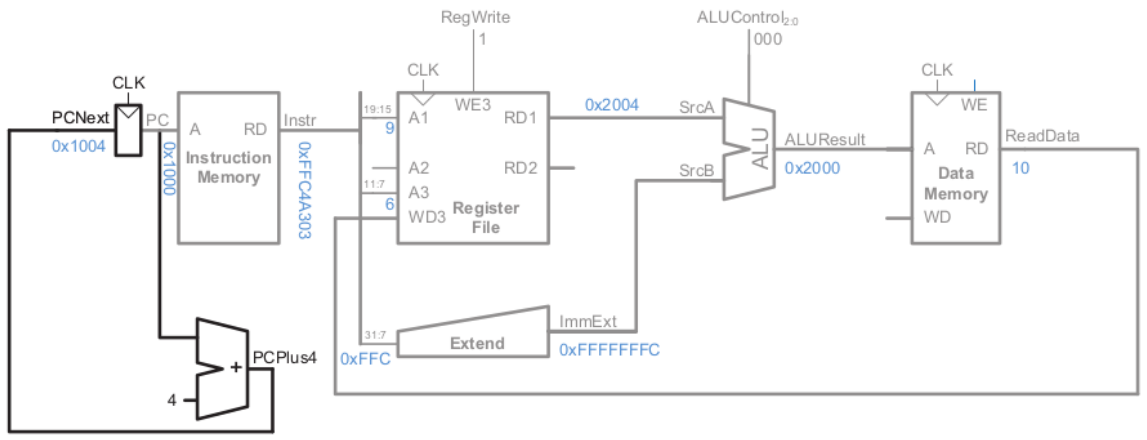
ALUResult is sent to the data memory as the address for the load instruction. The data is read from the data memory onto the ReadData bus, then written back to the destination register in the register file at the end of the cycle, as shown in Figure below. Port 3 of the register file is the write port.



Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

The destination register for the load instruction is specified in the rd field, Instr[11:7], which is connected to the port 3 address input, A3, of the register file. The ReadData bus is connected to the port 3 write data input, WD3, of the register file. A control signal called RegWrite is connected to the port 3 write enable input, WE3, and is asserted during a load instruction so that the data value is written into the register file.

The write takes place on the rising edge of the clock at the end of the cycle. While the instruction is being executed, the processor must compute the address of the next instruction, PCNext. Because instructions are 32 bits = 4 bytes, the next instruction is at PC + 4. Figure below shows that datapath uses another adder to increment the PC by 4. The new address is written into the program counter on the next rising edge of the clock. This completes the datapath for the load instruction.

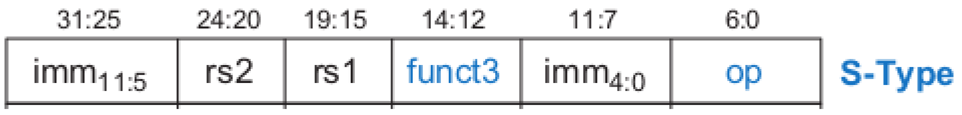


Address	Instruction	Type	Fields					Machine Language
			$imm_{11:0}$	rs1	f3	rd	op	
0x1000	L7: lw x6, -4(x9)	I	111111111100	01001	010	00110	0000011	FFC4A303

5.1 Designing Microarchitecture-III

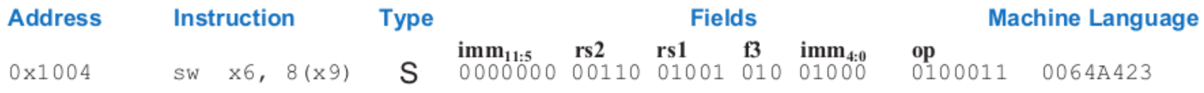
5.1.1 Introduction

Store instructions are S-type instructions. The **store word instruction**, sw, copies data from a register to memory. The register is not changed. The memory address is specified using a base/register pair. The name S-type is short for Store-type. S-type instructions use two register operands and one immediate operand. The figure below shows the S-type machine instruction format. The 32-bit instruction has five fields: op, rs1, rs2, funct3, and imm. The first four fields, op, rs1, rs2, and funct3 are like those of R-type instructions. The imm field holds the 12-bit immediate. The operation is determined by the opcode and funct3, highlighted in blue. The operands are specified in the three fields rs1, rs2, and imm. rs and imm are always used as source operands. rs2 is used as another source for store instructions.



31:25	24:20	19:15	14:12	11:7	6:0	
$imm_{11:5}$	rs2	rs1	funct3	$imm_{4:0}$	op	S-Type

The figure below shows an example of SW instruction represented in assembly code and machine code.



Address	Instruction	Type	$imm_{11:5}$	rs2	rs1	f3	$imm_{4:0}$	op	Machine Language
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

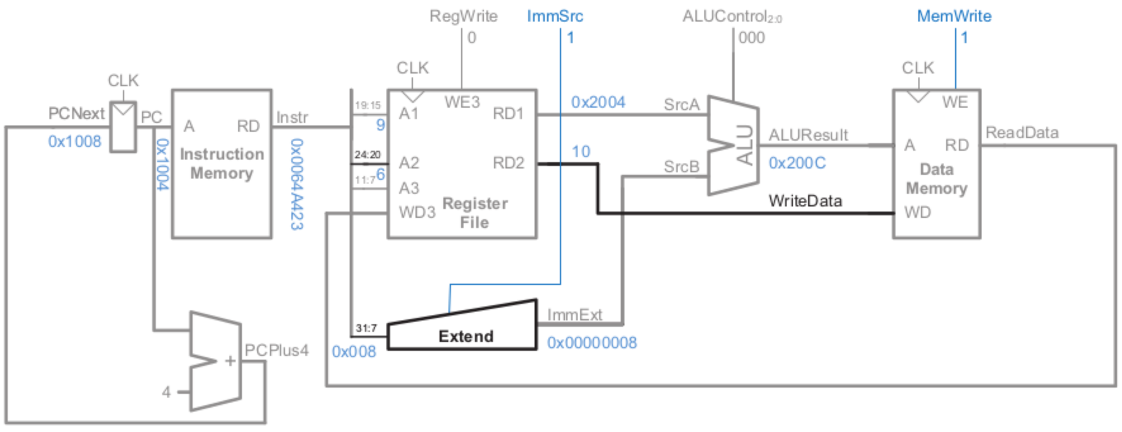
Like I-type instructions, S-type also has 12-bit immediates. The immediate in store instructions are split into two halves: the lower bits are stored in Instr[11:7] bit and the upper bits are stored in Instr[31:25]. First, we make the 12-bit immediate by concatenating the upper and lower part of immediate and then sign extend it for further use.

Example:
immediate = {Instr[31:25], Instr[11:7]}

5.1.2 DataPath for Store Word Instruction

In this section, we will extend the data path to Store instructions. Like the load instruction, the store instruction reads a base address from port 1 of the register file and sign-extends an immediate. The ALU adds the base address to the immediate to find the memory address. All of these functions are already supported by the datapath. The store instruction also reads a second register from the register file and writes it to the data memory. The figure below shows the new connections for this function. The register is specified in the rs2 field, Instr[24:20]. These bits of the instruction are connected to the second register file read port, A2.

The register value is read onto the RD2 port. It is connected to the write data port of the data memory. The write enable port of the data memory, WE are controlled by MemWrite. For a store instruction, MemWrite = 1, to write the data to memory; ALUControl = 000, to add the base address and offset; and RegWrite = 0, because nothing should be written to the register file. Note that data is still read from the address given to the data memory, but that this ReadData is ignored because RegWrite = 0.



Address	Instruction	Type	Fields						Machine Language
			imm$_{11:5}$	rs2	rs1	f3	imm$_{4:0}$	op	
0x1004	sw x6, 8(x9)	S	0000000	00110	01001	010	01000	0100011	0064A423

6.1 Designing Microarchitecture-IV

6.1.1 Introduction

The name R-type is short for register-type. R-type instructions use three registers as operands: two as sources, and one as a destination. Figure below shows the R-type machine instruction format. The 32-bit instruction has six fields: op, rs1, rs2, rd, funct3, and funct7. Each field is five or seven bits, as indicated. The operation the instruction performs is encoded in the three fields highlighted in blue: op (also called opcode or operation code) and funct3 and funct7 (also called the function). All R-type instructions have an opcode of 33. The specific R-type operation is determined by the funct field. The operands are encoded in the three fields: rs1, rs2, and rd. The first two registers, rs1 and rs2, are the source registers; rd is the destination register.

R-Type

31:25	24:20	19:15	14:12	11:7	6:0
funct7	rs2	rs1	funct3	rd	op
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

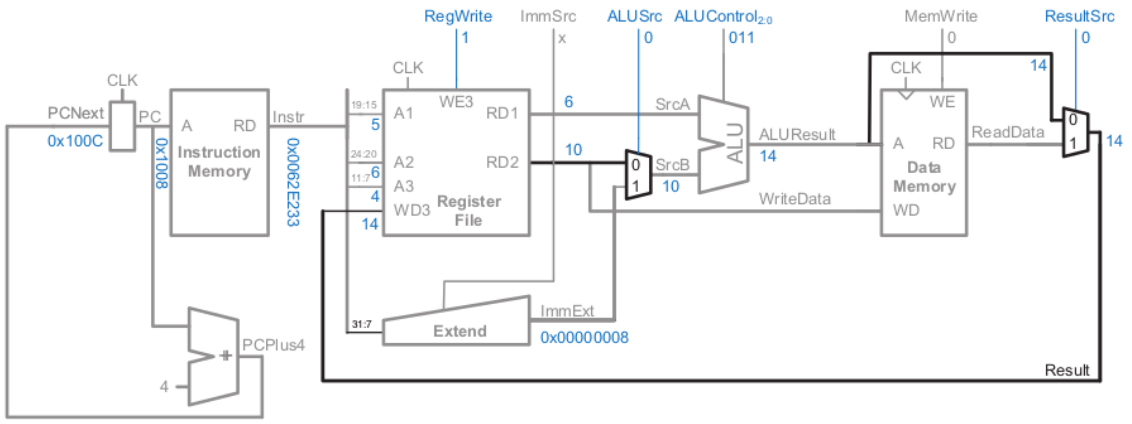
Figure below shows an example for R-Type instruction represented in assembly code and machine code.

Address	Instruction	Type	Fields						Machine Language
			funct7	rs2	rs1	f3	rd	op	
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233

6.1.2 DataPath for R-Type Instruction

In this section we will extend the data path to handle R-type instructions add, sub, and, or, and slt. All of these instructions read two registers from the register file, perform some ALU operation on them, and write the result back to a third register in the register file. They differ only in the specific ALU operation. Hence, they can all be handled with the same hardware, using different ALUControl signals.

Figure below shows the enhanced datapath handling R-type instructions. The register file reads two registers. The ALU performs an operation on these two registers. In previous labs we saw that the ALU always received its SrcB operand from the sign-extended immediate (ImmExt). Now, we add a multiplexer to choose SrcB from either the register file RD2 port or ImmExt. The multiplexer is controlled by a new signal, ALUSrc. ALUSrc is 0 for R-type instructions to choose SrcB from the register file; it is 1 for Load and store to choose ImmExt.



Address	Instruction	Type	funct7	rs2	rs1	f3	rd	op	Machine Language
0x1008	or x4, x5, x6	R	0000000	00110	00101	110	00100	0110011	0062E233

This principle of enhancing the datapath's capabilities by adding a multiplexer to choose inputs from several possibilities is extremely useful. Indeed, we will apply it twice more to complete the handling of R-type instructions. Previously, the register file always got its written data from the data memory. However, R-type instructions write the ALUResult to the register file. Therefore, we add another multiplexer to choose between ReadData and ALUResult. We call its output Result. This multiplexer is controlled by another new signal, ResultSrc. ResultSrc is 0 for R-type instructions to choose Result from the ALUResult; it is 1 for Load to choose ReadData. We don't care about the value of ResultSrc for store, because store does not write to the register file.