

EECS 151/251A **Spring 2021 Digital Design and Integrated** Circuits

- Instructor:
- John Wawrzynek

Lecture 14: RISC-V Part 2

Announcements

Virtual Front Row today, 3/4 □ Tony Kam Ben Tait **Robin Chu** □ Neil Kulkarni **Robert** Puccinelli □ HW6 posted (due Monday) Midterm Reminder □ Format TBD □ No HW next week

1:		2/25	Circuit Timing Part 2 (slides)(video)	
	7	3/2	RISC-V Microarchitecture and Implementation	Discussion 7
		3/4	RISC-V Part 2	
	8	3/9	Exam 1 Review	Discussion 8
		3/11	No Class – Exam 6–9PM	
	9	3/16	Power and Energy	Discussion 9
		3/18	Memory Blocks 1	
	10	3/23	Spring Recess	
		3/25	Spring Recess	



Implementir



- RISC-V Assembly Instruction, example beq rs1, rs2, label
 - stored in the immediate field(s)

example:

beq x1, x2, L1

- immediate
- But now immediate represents values -4096 to +4094 in 2-byte increments
- always zero, so no need to store it)

ng Branches										
Uses 12	the "B-type"	" instructio	on format 6							
funct3	imm[4:1]	imm[11]	opcode							
3	4	1	7							
le:										

if rs1 = rs2 pc \leftarrow pc + offset // offset computed by compiler/assembler and

• B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit

• The 12 immediate bits encode even 13-bit signed byte offsets (lowest bit of offset is







Review: Adding sw to datapath







Adding branches to datapath









Adding branches to datapath









Branch Comparator



- BrUn =1 selects unsigned comparison for BrLT, 0=signed

• BGE branch: A >= B, if !(A<B)



RISC-V Immediate Encoding

Instruction Encodings. inst[31:0]

31	30	25	24	21	20	19	1	5 14	12	11	8	7		6	0	
	funct7			rs2			rs1	fun	ct3		r	d		opcod	de	R-type
		$\operatorname{imm}[1]$	L:0]				rs1	fun	ct3		r	d		opcod	de	I-type
	imm[11:5]			rs2			rs1	fun	ct3		imm	[4:0]		opcod	de	S-type
imm[1	$[12] \mid \text{imm}$	[10:5]		rs2			rs1	fun	ct3	imr	n[4:1]	imm	[11]	opcod	de	B-type
	32-bit immediates produced, imm[31:0]															
31	30		20 19			12	11	10		5	4	1	1	0		
		- ins	st[31]					inst	[30 :	25	inst[24:21] ir	nst[20])]	I-imme
									-				-	-	-	
		- ins	st[31]					inst	30.	25]	inst	[11.8]	i	nst[7]		S-imme
		111.	oulor					11100	00.	20]	mot	[11.0]				
			1						[00]	0 F1	•			0	_	. .
]	nst[31]				1	nst[7]	inst	[30:	25	inst	[11:8]		0		B-imme
						→		0	nly	bit 7	7 of ir	nstru	ctior	n cha	nge	es role
oer bits	sign-ext	ended	l from	inst	[31] a	lway	'S	ir	nme	edia	te bet	twee	n S a	and B	-	

31	30	25	24	21	20	19	15	14	12	11	8	7	6		0
	funct7			rs2		rs1		func	ct3		\mathbf{rd}		C	opcode	e R-type
imm[11:0]					rs1		func	et3		rd		C	opcode	e I-type	
								-							
i	mm[11:5]			rs2		rs1		func	et3		imm[4	:0]	C	pcode	e S-type
• [4								0			[4]] .	[4	41		
imm[1	$2 \mid \text{imm}[1]$.0:5]		rs2		rsl		fund	ct3	imr	n[4:1] i	mm[1]	1] C	opcode	e B-type
<u>32-bit imn</u>							s p	rodu	ced	<u>, im</u>	m[31:0	<u>)]</u>			
1 3	30		20 1	9]	11		10		5	4	1		0	
		-in	st[31]					inst	[30:	25]	inst[2	4:21]	ins	t[20]	I-imme
									_	_	_	_			
	10.	-in	st[31]					inst	[30:	25]	inst[1	1:8]	in	st[7]	S-imme
			[]						[001]		[]		[.]	
	in	at [91	1			ingt	71	ingt	[20.	<u>95]</u>	ingt [1	1.0]		0	D imm
	— II.	Ist[31] —			Inst	[]	mst	[30:	20]	mst	1:0]		0	D-IIIIII
						→		0	nly	bit 7	7 of ins	truct	ion	chan	ges role
r bits	sign-exte	ndec	fror	n inst	[31] alv	ways		in	nme	edia ⁻	te betv	veen	S ar	nd B	
	31 imm[1 1 3	31 30 funct7 imm[11:5] imm[12] imm[1 1 30 	31 30 25 funct7 imm[11] imm[11:5] imm[12] imm[12] imm[10:5] 1 30 — in — in	31 30 25 24 funct7	31 30 25 24 21 funct7 rs2 imm[11:0] imm[11:5] rs2 imm[12] imm[10:5] rs2 ja2-k 32-k 1 30 20 - inst[31] - - inst[31] -	31 30 25 24 21 20 funct7 rs2 imm[11:0] imm[12] imm[10:5] rs2 imm[12] imm[10:5] rs2 30 20 19 1 30 20 19 - inst[31] - - inst[31] - r bits sign-extended from inst[31]	31 30 25 24 21 20 19 funct7 rs2 rs1 imm[11:0] rs1 imm[12] imm[10:5] rs2 rs1 32-bit immediate 1 30 20 19 12 11 — inst[31] — inst[rs1	31 30 25 24 21 20 19 15 funct7 rs2 rs1 imm[11:0] rs1 imm[12] imm[10:5] rs2 rs1 imm[12] imm[10:5] rs2 rs1 32-bit immediates p 1 30 20 19 12 11 — inst[31] — inst[7]	$\begin{array}{c c c c c c c c c c c c c c c c c c c $	31 30 25 24 21 20 19 15 14 12 funct7 rs2 rs1 funct3 imm[11:0] rs1 funct3 imm[12] imm[10:5] rs2 rs1 funct3 32-bit immediates produced 32-bit immediates produced 1 30 20 19 12 11 10 inst[31] inst[30: inst[30: inst[31] inst[30: 0nly r bits sign-extended from inst[31] always immediates 0nly	31 30 25 24 21 20 19 15 14 12 11 funct7 rs2 rs1 funct3 12 11 imm[11:0] rs2 rs1 funct3 12 11 imm[11:5] rs2 rs1 funct3 12 11 imm[12] imm[10:5] rs2 rs1 funct3 imm 30 20 19 12 11 10 5	31 30 25 24 21 20 19 15 14 12 11 8 funct7 rs2 rs1 funct3 rd imm[11:0] rs1 funct3 rd imm[11:5] rs2 rs1 funct3 imm[4 imm[12] imm[10:5] rs2 rs1 funct3 imm[4:1] i 30 20 19 12 11 10 5 4 - inst[31] - inst[30:25] inst[2] inst[2] - inst[31] - inst[7] inst[30:25] inst[1] - inst[31] - inst[7] inst[30:25] inst[1]	$\begin{array}{c c c c c c c c c c c c c c c c c c c $	31 30 25 24 21 20 19 15 14 12 11 8 7 6 funct7 rs2 rs1 funct3 rd c imm[11:0] rs1 funct3 rd c imm[11:5] rs2 rs1 funct3 imm[4:0] c imm[12] imm[10:5] rs2 rs1 funct3 imm[4:1] imm[11] c 30 20 19 12 11 10 5 4 1 1 30 20 19 12 11 10 5 4 1 - inst[31] - inst[30:25] inst[24:21] inst - inst[31] - inst[30:25] inst[11:8] inst - inst[31] - inst[7] inst[30:25] inst[11:8] inst - inst[7] inst[30:25] inst[11:8] - Only bit 7 of instruction immediate between S ar - immediate between S ar - <td>$\begin{array}{c c c c c c c c c c c c c c c c c c c$</td>	$\begin{array}{c c c c c c c c c c c c c c c c c c c $



8

Implementing **JALR** Instruction (I-Format)

31	20 19) 1	5 14 12	11 7	6	(
imm[11:0]		rs1	funct3	rd	opcode	
12		5	3	5	7	
offset[11:0]		base	0	dest	JALR	

- JALR rd, rs, immediate
 - Writes PC+4 to Reg[rd] (return address)
 - Sets PC = Reg[rs1] + offset
 - Uses same immediates as arithmetic and loads

no multiplication by 2 bytes





Review: Adding branches to datapath







Adding jalr to datapath



Adding jalr to datapath



Implementing jal Instruction



- JAL saves PC+4 in Reg[rd] (the return address) • Set PC = PC + offset (PC-relative jump) • Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart
- ±2¹⁸ 32-bit instructions
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost

Uses the "J-type" instruction format

19 12	11 7	6 0
imm[19:12]	rd	opcode
8	5	7
	dest	JAL





Adding jal to datapath







Adding jal to datapath



Single-Cycle RISC-V RV32I Datapath

Controller Implementation: Control logic works really well as a case statement... always @* begin op = instr[26:31];imm = instr[15:0]; ...reg dst = 1'bx; // Don't care reg write = 1'b0; // By default don't write • • • case (op) 6'b000000: begin reg write = 1; ... end • • •

Processor Pipelining

Program Execution Time

= (# instructions)(cycles/instruction)(seconds/cycle)

= # instructions x CPI x T_C

19

• T_C is limited by the critical path (lw)

Single-Cycle Performance

- Single-cycle critical path: $T_{c} = t_{q PC} + t_{mem} + max(t_{RFread}, t_{sext} + t_{mux}) + t_{ALU} +$ $t_{mem} + t_{mux} + t_{RFsetup}$
- In most implementations, limiting paths are: – memory, ALU, register file. $-T_{c} = t_{q PC} + 2t_{mem} + t_{RFread} + t_{mux} + t_{ALU} + t_{RFsetup}$

Pipelined Processor

- Use temporal parallelism
- Fetch
 - Decode
 - Execute
 - Memory
 - Writeback
- Add pipeline registers between stages

• Divide single-cycle processor into 5 stages:

<u>Single-Cycle vs. Pipelined Performance</u>

Single-Cycle

900	1000	1100	1200	1300	1400	1500	1600	1700	1800	190	00
Vrite Reg		1			1	1		I	Tir	ne (p	os)
	F Inst	etch ruction	De Re	ecode ad Reg	Exe Al	cute ₋U	Me Read	emory d / Writ	e R	rite eg	

Pipelined

mory I/Write		Write Reg					
ute U		Memory Read/Write			Write Reg		
Decode Read Reg		Exe AL	cute _U		Memory Read/Write		Write Reg

Single-Cycle and Pipelined Datapath

WriteReg must arrive at the same time as Result \bullet

Pipelined Control

Same control unit as single-cycle processor Control delayed to proper pipeline stage

Pipeline Hazards

Occurs when an instruction depends on results from previous instruction that hasn't completed.
 Types of hazards:

 Data hazard: register value not written back to register file yet
 Control hazard: next instruction not decided yet (caused by branches)

We need to design ways to avoid hazards, else we pay the price in CPI (cycles per instruction) and processor performance suffers.

Processor Pipelining

Deeper pipeline example.

IF1 IF2 **X1 X2** ID **X1** IF1 IF2 ID

Deeper pipelines => less logic per stage => high clock rate. But Deeper pipelines* => more hazards => more cost and/or higher CPI.

Cycles per instruction might go up because of unresolvable hazards.

Remember, Performance = # instructions X Frequency_{clk} / CPI

*Many designs included pipelines as long as 7, 10 and even 20 stages (like in the Intel Pentium 4). The later "Prescott" and "Cedar Mill" Pentium 4 cores (and their <u>Pentium D</u> derivatives) had a 31-stage pipeline.

How about shorter pipelines ... Less cost, less performance (but higher cost efficiency)

- WB **M1 M2**
- **X2 M1** WB **M2**

3-Stage Pipeline

3-Stage Pipeline (used for FPGA/ASIC project)

The blocks in the datapath with the greatest delay are: IMEM, ALU, and DMEM. Allocate one pipeline stage to each:

Use PC register as address to IMEM and retrieve next *instruction. Instruction gets* stored in a pipeline register, also called "instruction register", in this case.

Most details you will need to work out for yourself. Some details to follow ... In particular, let's look at hazards.

Use ALU to compute result, memory address, or branch target address.

Access data memory or I/O device for load or store. Allow for setup time for register file write.

add	x5, x3, x4	
add	x7, x6, x5	

reg 5 value needed here!

The fix:

Selectively forward ALU result back to input of ALU.

Data Hazard

Need to add mux at input to ALU, add control logic to sense when to activate. Check reference for details.

lw x5, offset(x4)	
add x7, x6, x5	

value needed here!

The fix: Delay the dependent instruction by one cycle to allow the load to complete, send the result of load directly to the ALU (and to the regfile). No delay if not dependent!

lw x5, offset(x4

add x7, x6, x5

add x7, x6, x5

Load Hazard

Memory value known here. It is written into the regfile on this edge.

4)	/	X	Μ		
		1	пор	пор	
			1	X	Μ

beq x1, x2, L1	X	M		
add x5, x3, x4	I	X	Μ	
add x6, x1, x2		I	X	Μ
L1: sub x7, x6, x5				X

but needed here!

Several Possibilities:* The fix: 1. Always delay fetch of instruction after branch 3. and correct later if wrong.

- 1.
- 3.
- * MIPS defines "branch delay slot", RISC-V doesn't

Control Hazard

branch address ready here

2. Assume branch "not taken", continue with instruction at PC+4, and correct later if wrong.

Predict branch taken or not based on history (state)

Simple, but all branches now take 2 cycles (lowers performance) Simple, only some branches take 2 cycles (better performance) Complex, very few branches take 2 cycles (best performance)

Branch address ready at end of X stage:

- If branch "not taken", do nothing. •
- lacksquare

<u>bneq</u> x1, x1, L1	Χ	Μ		
add x5, x3, x4		Χ	Μ	
add x6, x1, x2			Χ	Μ
L1: sub x7, x6, x5				Χ

<u>beq</u> x1, x1, L1	Χ	Μ		
add x5, x3, x4		nop	nop	
L1: sub x7, x6, x5			Χ	Μ

Control Hazard

If branch "taken", then kill instruction in I stage (about to enter X stage) and fetch at new target address (PC)

Not taken

Taken

EECS151 Project CPU Pipelining Summary

□ Pipeline rules:

- Writes/reads to/from DMem are clocked on the leading edge of the clock in the "M" stage
- Writes to RegFile at the end of the "M" stage
- Instruction Decode and Register File access is up to you.
- □ Branch: predict "not-taken"
- Load: 1 cycle delay/stall on dependent instruction
- Bypass ALU for data hazards
- More details in upcoming spec

