دانشگاه صنعتی امیرکبیر

دانشکده مهندسی کامپیوتر و فناوری اطلاعات

آزمایشگاه معماری کامپیوتر

معماری MIPS پایه

بهار ۱۳۹۵

در پروژه‌ی نهایی این درس هدف جمع‌بندی مطالب ارائه‌شـده در طول ترم و طراحی یک پردازنده‌ی AUT-MIPS است. در این پروژه یـک پردازنده‌ی ۱۶ بیتی با ویژگی‌هایی که در ادامه می‌آید باید طراحی گردد. این پردازنده نســـبت به پردازنده‌ی MIPS تغییراتی کرده اســت. طول دســتورات این پردازنده ۱۶ بیت بوده و به همین دلیل در این پردازنده PC به جای آنکه با ۴ جمع شود، با ۲ جمع می‌شود (بنابراین حافظه در اینجا طول ۸ بیتی خواهد داشت و در صورتی که از حافظه ۱۶ بیتی استفاده کنید برای محاسبه مقدار PC باید PC+1 شود.). همچنین طول ثبات PC ۱۶بیت است. دستورات این پردازنده به سه دسته‌ی کلی R-Type، I-Type و J-Type تقسیم می‌شود که در ادامه ویژگی‌های هرکدام ذکر می‌شود.

۱- R-Type: این دستور همانند دستور R-Type در پردازنده‌ی MIPS است که دو ثبات مبداء و یک ثبات مقصد دریافت می‌کند. ساختار دستور و جدول opcode این دستور در ادامه آمده است:

4 bits	4 bits	4 bits	4 bits
operation code (15-12)	Destination (11-8)	Source1 (7-4)	Source2 (3-0)

شکل ۱. ساختار دستور R-Type

opcode	Function
0000	ADD
0001	SUB
0010	AND
0011	OR
0100	MUL
0101	DIV
0110	XOR
0111	NOR

جدول۱: مقدارopcode برای دستورات R-Type

۲- I-Type: برای دستورات Addi , Subi چهار بیت اول (بیت‌های ۱۲ الی ۱۵) opcode را مشخص می‌کند. چهار بیت ۸ الی ۱۱ شماره مقصد را مشخص می‌کند. در واقع عملیات خواسته شده روی مقدار این ثبات با مقدار immediate انجام شده و نتیجه در همین ثبات ذخیره می‌شود. ۸ بیت ۰ الی ۷ مقدار immediate را مشخص می‌کند. این مقدار ۸ بیتی sign extend می‌شود و به یک مقدار ۱۶ بیتی تبدیل شده و بعد با محتوای ثبات عملیات مربوط انجام می‌شود. مقادیر opcde برای دستورات Addi, Subi در جدول ۲ آمده است.

4 bits	4 bits	8 bits
Opcode (15-12)	Register1 (11-8)	Immediate (7-0)

شکل ۲. قالب دستورات I-Type

opcode	Function
1000	Addi
1001	Subi

جدول ۲. مقادیر opcode برای دستورات Addi , Subi

برای این پردازنده دستورات پرش جز دستورات I-type هستند. دستورات پرش محتوای دو ثبات را با یکدیگر مقایسه می‌کنند و براساس نوع مقایسه، اگر نتیجه درست باشد پرش انجام می‌شود و در غیر این صورت انجام نمی‌شود. آدرس پرش به این صورت محاسبه می‌شود که مقدار۸ بیتی به مقدار ۱۶بیتی sign extend می‌شود سپس در دو ضرب می‌گردد و بعد با PC+2 جمع می‌شود.

PC+2+(sign-extend(immediate)*2)

opcode	Fuction
1010	beq(branch if src1=scr2)
1011	bgt(branch if src1>src2)

جدول ۳. مقادیر opcode برای دستورات پرش

برای این پردازنده دستورات Load, Store به صورت زیر تعریف می‌شود. دستور Load یک داده ۱۶ بیتی را از حافظه می‌خواند و Store یک داده ۱۶ بیتی را در حافظه می‌نویسد. در این پردازنده مانند MIPS از آدرس‌دهی displacement (محتوای ثبات +offset) برای محاسبه آدرس استفاده می‌شود. برای دستورات Load چهاربیت اول یعنی بیت‌های ۱۲ الی ۱۵ opcode را مشخص می‌کند. چهار بیت بعد یعنی بیت‌های ۸ الی ۱۱ شماره ثبات پایه را مشخص می‌کند. چهاربیت ۴ الی ۷ شماره ثبات مقصد را مشخص می‌کند. ۴ بیت آخر بیت‌های ۰ الی ۳ مقدار offset را مشخص می‌کند که باید sign extend شود و بعد با ثبات پایه جمع شود و آدرس را تولید کند.

برای دستورات Store چهاربیت اول (۱۲-۱۵) opcode را مشخص می‌کند. چهار بیت بعد یعنی بیت‌های ۸ الی ۱۱ شماره ثبات پایه را مشخص می‌کند. چهاربیت ۴ الی ۷ شماره ثبات مقصد را مشخص می‌کند. ۴ بیت آخر بیت‌های ۰ الی ۳ مقدار offset را مشخص می‌کند که باید sign extend شود و بعد با ثبات پایه جمع شود و آدرس را تولید کند. مقدار offset یک مقدار علامت دار می‌باشد.

مقادیر opcode برای دستورات Load, Store در جدول ۴ آمده است.

opcode	Fuction
1101	Load
1110	Store

جدول ۴. مقادیر opcode برای دستورات Load, Store

۳- J-Type: دستور jmp از نوع J-Type می‌باشد. ماشین کد دستورات J-Type در شکل ۳ آورده شده است. در این دستور ۱۲ بیت پایین یعنی بیتهای ۰ الی ۱۱ در ۲ ضرب شده تبدیل به یک مقدار ۱۳ بیتی می‌شود و سپس سه بیت بالای PC به ابتدای آن اضافه می‌شود تا یک مقدار ۱۶ بیتی ایجاد گردد و سپس در PC نوشته می‌شود.

Opcode دستور jmp برابر با ۱۱۰۰می‌باشد.

PC<- PC[15…13] && (instr[11…0]) && “0”

4 bits	12 bits
Opcode (1100)	Jump address

شکل ۳. قالب دستورات J-Type

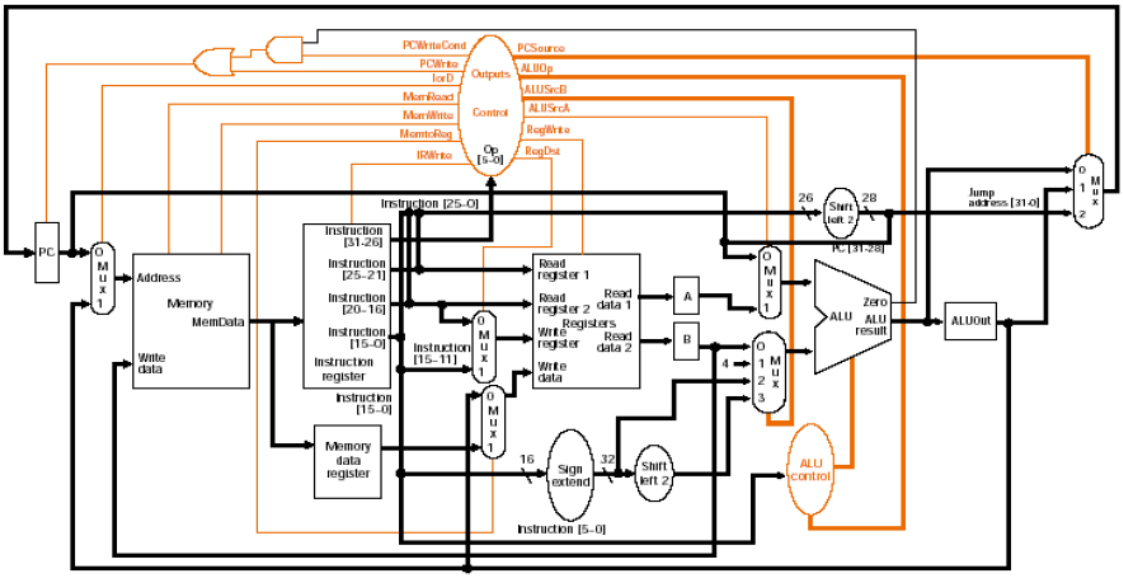
دستور نهایی دستور reset است که opcode آن برابر با ۱۱۱۱ می‌باشد و مقدار PC و تمام رجیسترها را صفر قرار می‌دهد.

موفق باشید

راهنمای هریک از مراحل اجرای دستورات

Multi-cycle datapath: instruction execution

Breaking instruction execution into multiple clock cycles:
Balance amount of work done in each cycle (minimizes the cycle time)
Each step contains at most one:
Register access
Memory access
ALU operation
Any data values which are needed in a later clock cycle are stored in a register
Major state elements: PC, register file, memory
Temporary registers written on every cycle: A data, B data, MDR, ALUOut
Temporary register with write control: IR
Note that we can read the current value of a destination register:
New value doesn't get written until next clock cycle
Multiple operations can occur in parallel during same clock cycle
Read instruction and increment PC
Other operations occur in series during separate clock cycles
Reading or writing standalone registers (PC, A data, B data, etc.) done in 1 cycle
Register file access requires additional cycle: more overhead for access and control
Instruction execution steps
1. Fetch instruction from memory and compute address of next sequential instruction
2. Instruction decode and register fetch
3. R-type execution, memory address computation, or branch
4. Memory access or R-type instruction completion
5. Memory read completion



Multi-cycle datapath: instruction fetch

1. Fetch instruction from memory and compute address of next instruction

Operation:
IR = Memory[PC];
PC = PC + 2;

Control signals needed

MemRead, IRWrite asserted
IorD set to 0 to select PC as address source

Increment PC by 2:
- ALUSrcA = 0: PC to ALU
- ALUSrcB = 01: 2 to ALU
- ALUOp = add

Store PC back
- PCSource = 00: ALU result
- PCWrite = 1

The memory access and PC increment can occur in parallel. Why?
- Because the PC value doesn't change until the next clock cycle!

Where else is the incremented PC value stored?
- ALUOut

Does this have any other effect? No

Multi-cycle datapath: decode

2. Instruction decode and register fetch

What do we know about the type of instruction so far? Nothing!

So, we can only perform operations which apply to all instructions, or do not conflict with the actual instruction

What can we do at this point?
- Read the registers from the register file into A and B
- Compute branch address using ALU and save in ALUOut

But, what if the instruction doesn't use 2 registers, or it isn't a branch?
- No problem; we can simply use what we need once we know what kind of instruction we have
- This is why having a regular instruction pattern is a good idea

Is this inefficient?
- It does use up a little more power and generate some heat, but it doesn't cost any TIME
- In fact, it means that the entire instruction can be executed in fewer clock cycles

Operation:
- A = Reg[IR[15-12]];
- B = Reg[IR[11-8]];
- ALUOut = PC + sign_extend (IR[7-0]) << 2;

What are the control signals to determine whether to write registers A and B?
- There aren't any! We can read the register file and store A and B on EVERY clock cycle.

Branch address computation:
- ALUSrcA = 0: PC to ALU
- ALUSrcB = 11: sign-extended/shifted immediate to ALU
- ALUOp = add

These operations occur in parallel.

Multi-cycle datapath: ALU, memory address, or branch

3. R-type execution, memory address computation, or branch

ALU operates on the operands, depending on class of instruction

Memory reference:
- ALUOut = A + sign_extend (IR[7-0]);
- Operation: ALU creates memory address by adding operands
- Control signals
  - ALUSrcA = 1: register A

ALUSrcB = 10: sign-extension unit output
ALUOp = add

Arithmetic-logical operation (R-type):

ALUOut = A op B;
Operation:
ALU performs operation specified by function code on values in registers A, B
(Where did these operands come from?
They were read from the register file on the previous cycle.)
Control signals
ALUSrcA = 1: register A
ALUSrcB = 00: register B
ALUOp = 10: use function code bits to determine ALU control

Branch:

If (A == B) PC = ALUOut;

Operation:

ALU compares A and B. If equal, Zero output signal is set to cause branch, and PC is updated with branch address

Control signals

ALUSrcA = 1: register A
ALUSrcB = 00: register B
ALUOp = 01: subtract
PCWriteCond = 1: update PC if Zero signal is 1
PCSource = 01: ALUOut

(What is in ALUOut, and how did it get there?
It's the branch address calculated from the previous cycle, NOT the result of A - B.
Why not? Because ALUOut is updated at the END of each cycle.)
Note that PC is actually updated twice if the branch is taken:
Output of the ALU in the previous cycle (instruction decode/register fetch), From ALUOut if A and B are equal
Could this cause any problems? No, because only the last value of PC is used for the next instruction execution.

Jump:

PC = PC[15-13] || (IR[11-0] << 2);
Operation:
PC is replaced by jump address.
(Upper 3 bits of PC are concatenated with 26-bit address field of instruction shifted left by 2 bits)

Control signals

PCSource = 10: jump address
PCWrite = 1: update PC

(Where did the jump address come from?
Output of shifter concatenated with upper 3 bits of PC.)

Multi-cycle datapath: memory access/ALU completion

4. Memory access or R-type instruction completion

Load or store: accesses memory
Arithmetic-logical operation writes result to register

Memory reference

MDR = Memory[ALUOut]; or
Memory[ALUOut] = B;

Operation:

If operation is load, word from memory is put into MDR.

If operation is store, memory location is written with value from register B.

(Where does memory address come from?

It was computed by ALU in previous cycle.

Where does register B value come from?

It was read from register file in step 3 and also in step 2.)

Control signals

MemRead = 1 (load) or

MemWrite = 1 (store)

IorD = 1: address from ALU, not PC

What about MDR?

It's written on every clock cycle.

Arithmetic-logical operation

Reg[IR[11-8]] = ALUOut;

Operation:

ALUOut contents are stored in result register.

Control signals

RegDst = 1: use $rd field from IR for result register

RegWrite = 1: write the result register

MemtoReg = 0: write from ALUOut, not memory data

Multi-cycle datapath: memory read completion

5. Memory read completion

Value read from memory is written back to register

Reg[IR[20-16]] = MDR;

Operation:

Write the load data from MDR to target register $rt

Control signals

MemtoReg = 1: write from MDR

RegWrite = 1: write the result register

RegDst = 0: use $rt field from IR for result register