Multi-cycle datapath

Multi-cycle implementaion: break up instructions into separate steps

Each step takes a single clock cycle

Each functional unit can be used more than once in an instruction,

as long as it is used in different clock cycles

Reduces amount of hardware needed

Reduces average instruction time

Differences with single-cycle

Single memory for instructions and data

Single ALU (no separate adders for PC or branch calculation)

Extra registers added after major functional units to hold results between clock cycles

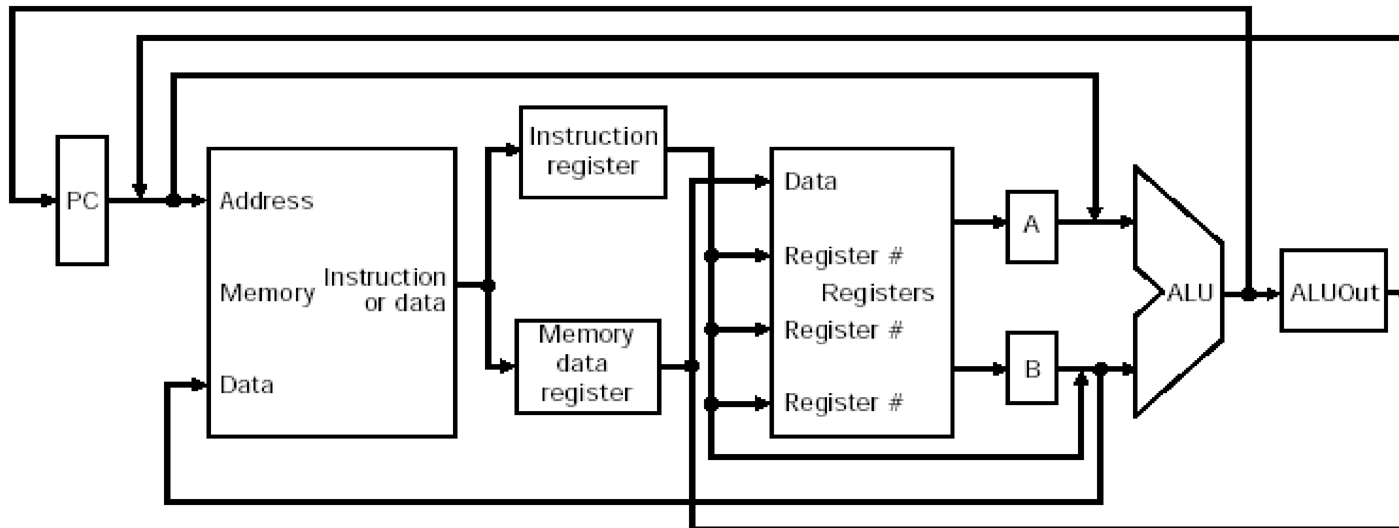


Fig. 5.30

Note that data needed in a later instruction must be in one of the programmer-visible registers or memory

Assume each clock cycle includes at most one of:

- **Memory access**
- **Register file access (2 reads OR 1 write)**
- **ALU operation**

Any data produced from 1 of these 3 functional units must be stored between cycles

- **Instruction register: contains current instruction**
- **Memory data register: data from main memory**
- **Why 2 separate registers? Because both values are needed simultaneously**

Register output A, B

- **2 operand values read from register file**

ALUOut

- **Output from ALU**
- **Why is this needed? Because we are combining adders into the ALU, so we need to select where the output goes (register file or memory)**

All these registers except IR hold data only between consecutive clock cycles, so don't need write control signal

What else do we need?

Because functional units are used for multiple purposes:

- **More MUXes**
- **More inputs for existing MUXes**

Multi-cycle datapath

MUX example 1:

One memory is used for instructions and data, so we need a MUX to select between:

- **PC (instruction)**
- **ALUout (data)**
- **for address to access in memory**

Where else? (Hint: Consider ALU)

MUX example 2:

One ALU is used to perform all arithmetic and logic operations, so we need a MUX to select first operand between

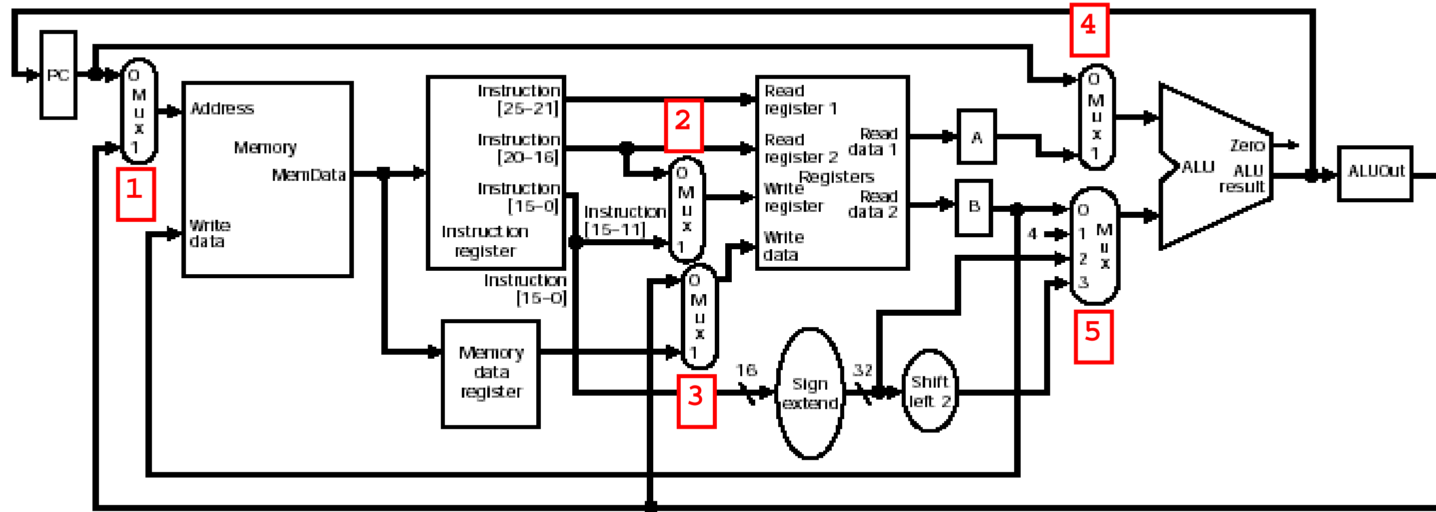
- **PC**
- **Data register A**

Also, for second operand:

- **Data register B**
- **Sign-extended immediate**
- **Sign-extended/shifted immediate (offset for branch)**
- **Constant 4 (incrementing PC)**

Multi-cycle datapath

Datapath with MUXes for selection:



MUX 1: select between PC and ALUOut for memory address **Fig. 5.31**

MUX 2: select between $rt and $rd for destination (write) register address

MUX 3: select between ALUOut and memory data for write data input to register file

MUX 4: select between PC and register data A for first operand input to ALU

MUX 5: select between

register data B

constant 4

sign-extended immediate

sign-extended, shifted immediate

for second operand input to ALU

Multi-cycle datapath

Control signals needed to select inputs, outputs

Need write control:

- Programmer-visible units
  - PC, memory, register file
- IR: needs to hold instruction until end of execution

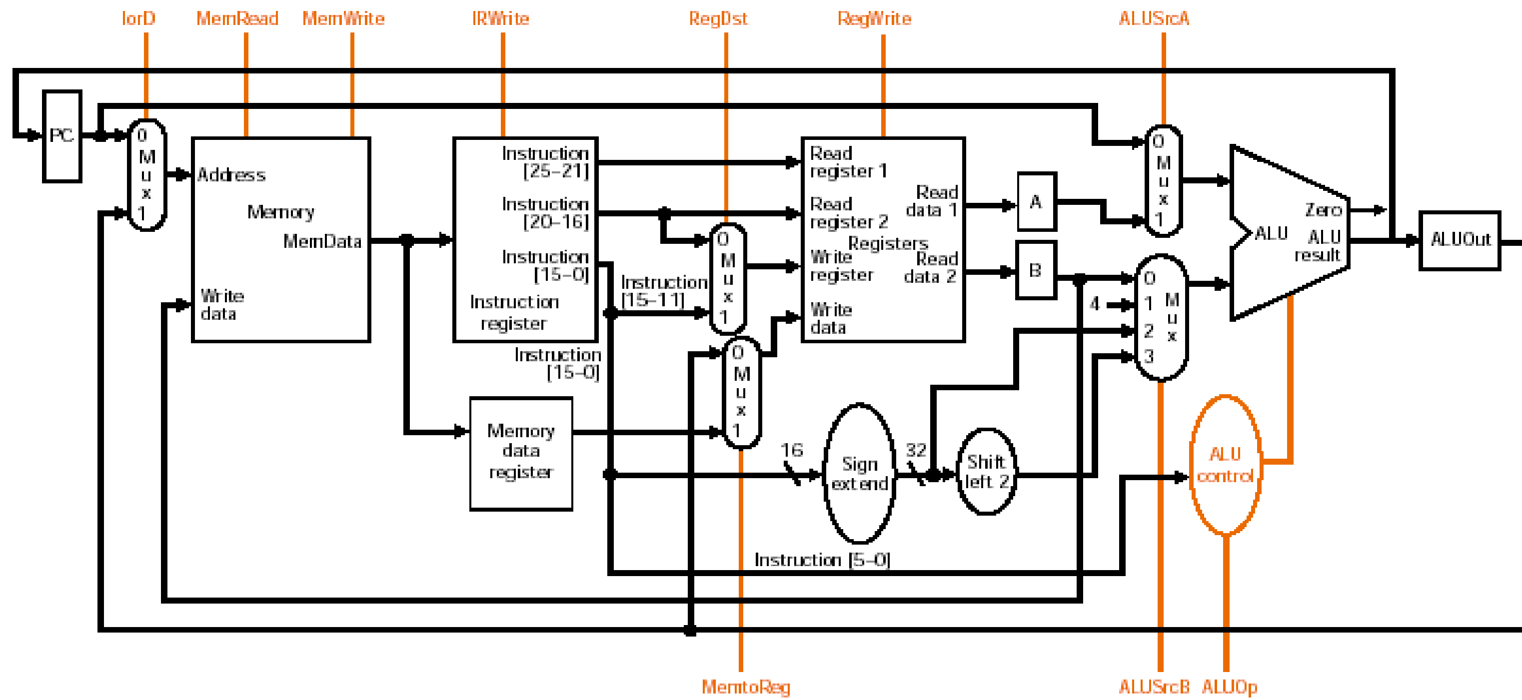
Need read control:

- memory

ALU Control: can use same control as single-cycle

MUXes: single or double control lines (depending on 2 or 4 inputs)

Multi-cycle datapath: control signals

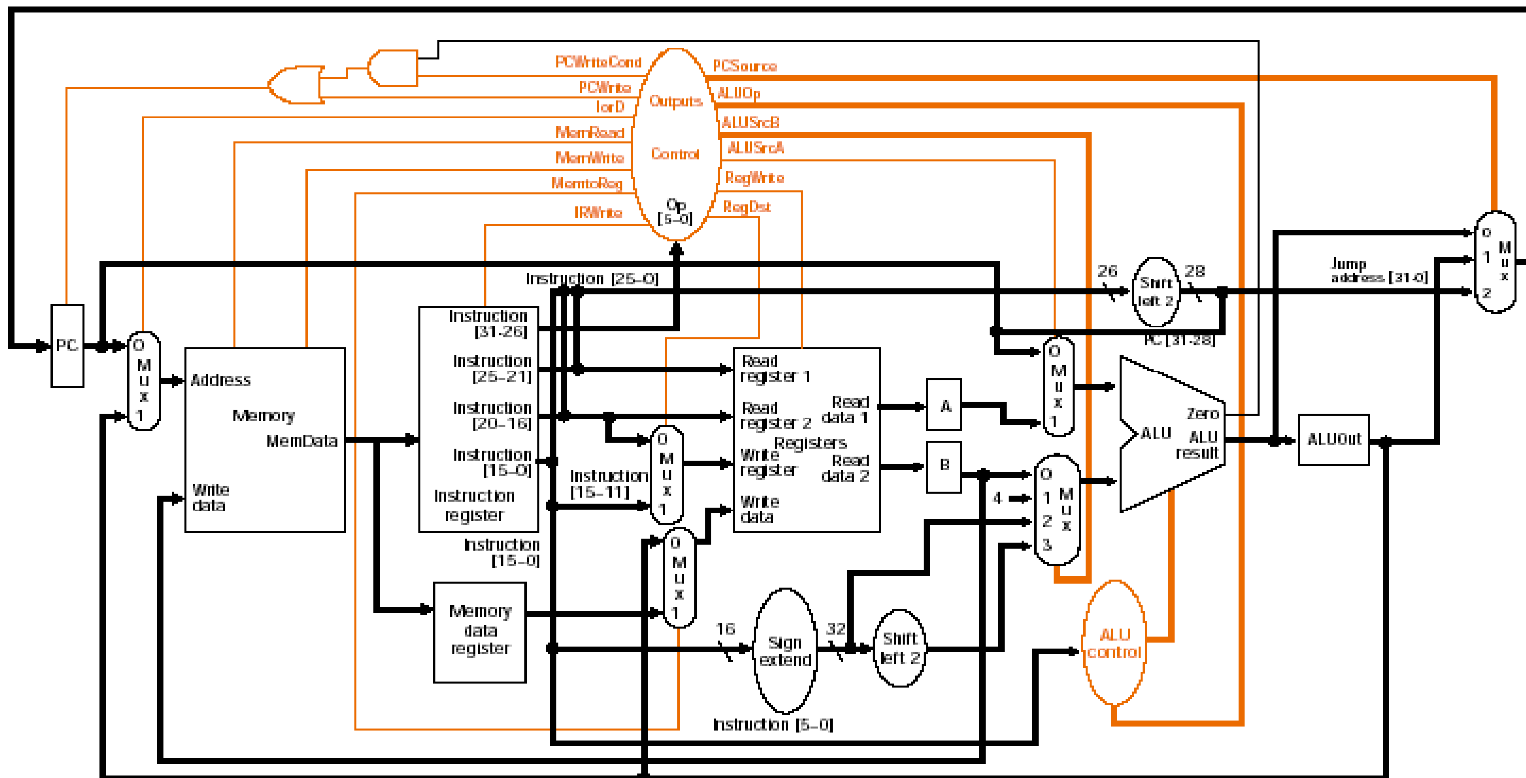


New control signals **Fig. 5.32**

- **IorD: selects PC (instruction) or ALUOut (data) for memory address**
- **IRWrite: updates IR from memory (when?)**
- **ALUSrcA: control to select PC or reg A (read data 1 from register file)**
  - **output is first operand for ALU**
- **ALUSrcB: control to select second operand for ALU among 4 inputs:**
  - **0: reg B (read data 2 from register file)**
  - **1: constant 4**
  - **2: sign-extended immediate from instruction**
  - **3: above value shifted left by 2**

Multi-cycle datapath: control signals

What else is needed? Branches and jumps



Possible sources for PC value: **Fig. 5.33**

- **(PC + 4) directly from ALU**
- **ALUout: result of branch calculation**
- **Result of concatenation of left-shifted 26 bits with upper 4 bits of PC (jump)**

Note that the PC is updated both unconditionally and conditionally,

- **so 2 control signals are needed**

PCWriteCond: ANDed with ALU Zero to control PC update for branch

- **This result is ORed with PCWrite**

PCSource: controls MUX to select input to PC

- **0: ALU result**
- **1: ALUOut**
- **2: Jump address**
- **Why do we need both 0 and 1 inputs?**

Control signals are listed in Fig. 5.34

Multi-cycle datapath: instruction execution

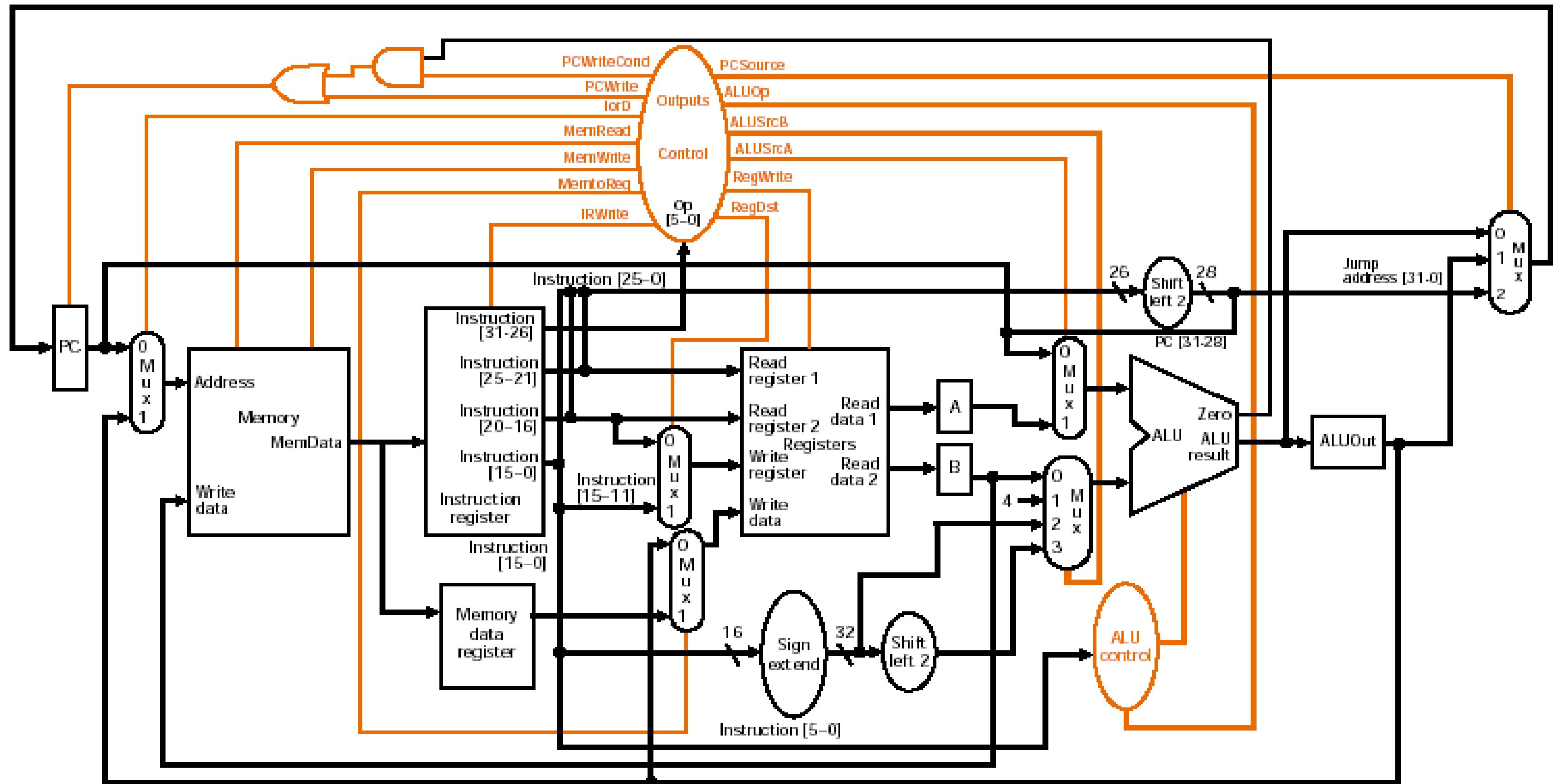
Breaking instruction execution into multiple clock cycles:

- **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

Fig. 5.33

Operation:

IR = Memory[PC];

PC = PC + 4;

Control signals needed

MemRead, IRWrite asserted

IorD set to 0 to select PC as address source

Increment PC by 4:

ALUSrcA = 0: PC to ALU

ALUSrcB = 01: 4 to ALU

ALUOp = 00: 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[25-21]];
- B = Reg[IR[20-16]];
- ALUOut = PC + sign_extend (IR[15-0]) << 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 = 00: 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[15-0]);**
- **Operation: ALU creates memory address by adding operands**
- **Control signals**
  - **ALUSrcA = 1: register A**
  - **ALUSrcB = 10: sign-extension unit output**
  - **ALUOp = 00: 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[31-28] || (IR[25-0] << 2);
Operation:
PC is replaced by jump address.
(Upper 4 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 4 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[15-11]] = 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

Multi-cycle datapath: summary

<table>
<tr><th>Step name</th><th>Action for R-type instructions</th><th>Action for memory-reference instructions</th><th>Action for branches</th><th>Action for jumps</th></tr>
<tr><td>Instruction fetch</td><td colspan="4">IR = Memory[PC]
PC = PC + 4</td></tr>
<tr><td>Instruction decode/register fetch</td><td colspan="4">A = Reg [IR[25–21]]
B = Reg [IR[20–16]]
ALUOut = PC + (sign-extend (IR[15–0]) << 2)</td></tr>
<tr><td>Execution, address computation, branch/ jump completion</td><td>ALUOut = A op B</td><td>ALUOut = A + sign-extend (IR[15–0])</td><td>if (A == B) then PC = ALUOut</td><td>PC = PC [31–28] || (IR[25–0]<<2)</td></tr>
<tr><td>Memory access or R-type completion</td><td>Reg [IR[15–11]] = ALUOut</td><td>Load: MDR = Memory[ALUOut]
or
Store: Memory [ALUOut] = B</td><td></td><td></td></tr>
<tr><td>Memory read completion</td><td></td><td>Load: Reg[IR[20–16]] = MDR</td><td></td><td></td></tr>
</table>

Fig. 5.35

Summary of execution steps

Instruction fetch, decode, register fetch same for all instructions

This document was created with Win2PDF available at http://www.daneprairie.com.
The unregistered version of Win2PDF is for evaluation or non-commercial use only.