Chapter 4
The Processor (Part 2)

王振傑 (Chen-Chieh Wang)
ccwang@mail.ee.ncku.edu.tw

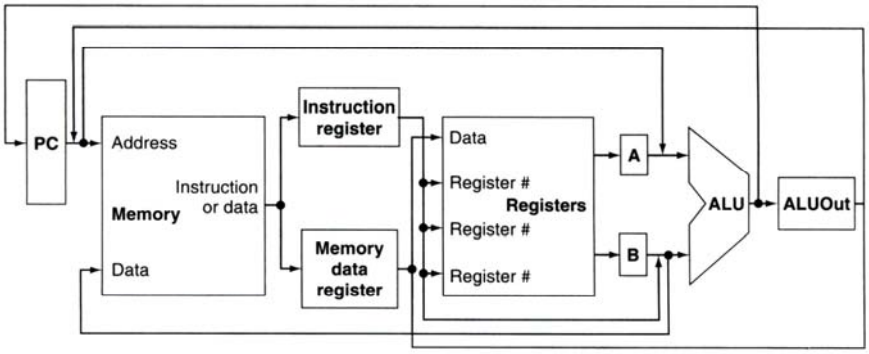
Department of Electrical Engineering, Feng-Chia University

Outline

- A Multicycle Implementation
- Mapping Control to Hardware (D.3, D.4)
- Microprogramming: Simplifying Control Design (D.5)
- Concluding Remarks

Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area
- One Solution:
  - use a "smaller" cycle time
  - have different instructions take different numbers of cycles
  - a "multicycle" datapath:



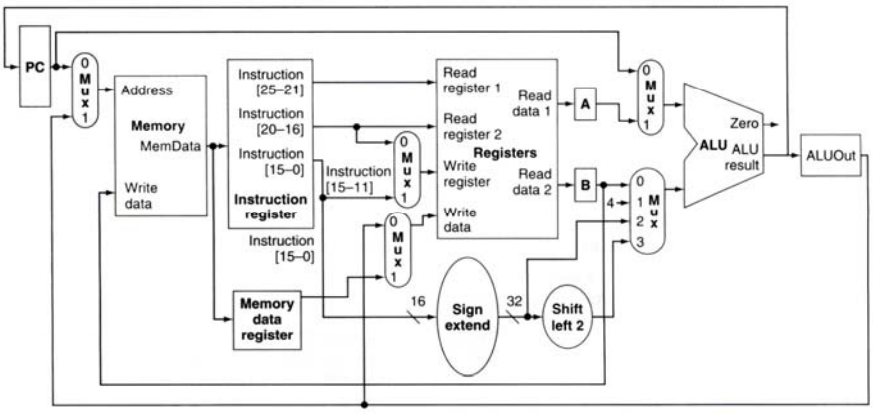
Department of Electrical Engineering, Feng-Chia University

Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Our control signals will not be determined directly by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine for control
  - This is a sequential circuit.

Multicycle Approach

- Break up the instructions into steps, each step takes a cycle
  - balance the amount of work to be done
  - restrict each cycle to use only one major functional unit
- At the end of a cycle
  - store values for use in later cycles (easiest thing to do)
  - introduce additional "internal" registers



Instructions from ISA perspective

- Consider each instruction from perspective of ISA.
- Example:
  - The add instruction changes a register.
  - Register specified by bits 15:11 of instruction.
  - Instruction specified by the PC.
  - New value is the sum ("op") of two registers.
  - Registers specified by bits 25:21 and 20:16 of the instruction

```
Reg[Memory[PC][15:11]] <=
        Reg[Memory[PC][25:21]] op
        Reg[Memory[PC][20:16]]
```

  - In order to accomplish this we must break up the instruction.
    (kind of like introducing variables when programming)

Breaking down an instruction

- ISA definition of arithmetic:

```
Reg[Memory[PC][15:11]] <=
          Reg[Memory[PC][25:21]] op
          Reg[Memory[PC][20:16]]
```

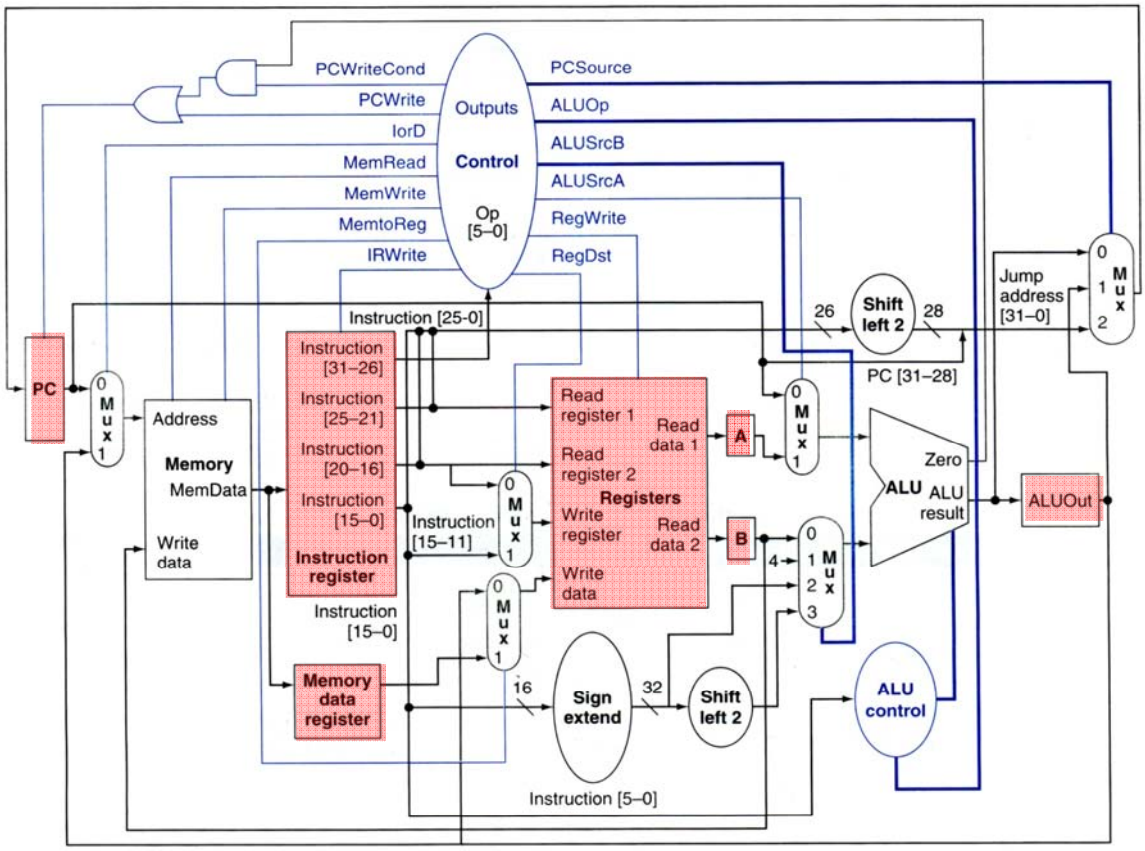
- Could break down to:
  - `IR <= Memory[PC]`
  - `A <= Reg[IR[25:21]]`
  - `B <= Reg[IR[20:16]]`
  - `ALUOut <= A op B`
  - `Reg[IR[15:11]] <= ALUOut`

- We forgot an important part of the definition of arithmetic!
  - `PC <= PC + 4`

Department of Electrical Engineering, Feng-Chia University

Idea behind multicycle approach

- We define each instruction from the ISA perspective (do this!)
- Break it down into steps following our rule that data flows through at most one major functional unit (e.g., balance work across steps)
- Introduce new registers as needed (e.g, A, B, ALUOut, MDR, etc.)
- Finally try and pack as much work into each step (avoid unnecessary cycles) while also trying to share steps where possible (minimizes control, helps to simplify solution)
- Result: Our book's multicycle Implementation!



Department of Electrical Engineering, Feng-Chia University

Five Execution Steps

1. Instruction Fetch
2. Instruction Decode and Register Fetch
3. Execution, Memory Address Computation, or Branch Completion
4. Memory Access or R-type instruction completion
5. Write-back step

INSTRUCTIONS TAKE FROM 3 - 5 CYCLES!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR <= Memory[PC];
PC <= PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Department of Electrical Engineering, Feng-Chia University

Step 2 : Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A <= Reg[IR[25:21]];
B <= Reg[IR[20:16]];
ALUOut <= PC + (sign-extend(IR[15:0]) << 2);
```

- We aren't setting any control lines based on the instruction type (we are busy "decoding" it in our control logic)

Step 3 : Execution (instruction dependent)

- ALU is performing one of three functions, based on instruction type
- Memory Reference:

```
ALUOut <= A + sign-extend(IR[15:0]);
```

- R-type:

```
ALUOut <= A op B;
```

- Branch:

```
if (A==B) PC <= ALUOut;
```


Step 4 : R-type or memory-access

- Loads and stores access memory

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

- R-type instructions finish

```
Reg[IR[15:11]] <= ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Step 5 : Write-back step

- Write-back

```
Reg[IR[20:16]] <= MDR;
```

Which instruction needs this?

Summary

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR <= Memory[PC] PC <= PC + 4			
Instruction decode/ register fetch	A <= Reg[IR[25:21]] B <= Reg[IR[20:16]] ALUOut <= PC + (sign-extend(IR[15:0]) << 2)			
Execution, address computation, branch/jump completion	ALUOut <= A op B	ALUOut <= A + sign-extend(IR[15:0])	if (A==B) PC <= ALUOut	PC <= {PC[31:28], IR[25:0], 2'b00}
Memory access or R-type completion	Reg[IR[15:11]] <= ALUOut	Load : MDR <= Memory[ALUOut] or Store : Memory[ALUOut] <= B		
Memory read completion		Load: Reg[IR[20:16]] <= MDR		

Simple Questions

- How many cycles will it take to execute this code?

```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label        #assume not
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place?

Department of Electrical Engineering, Feng-Chia University

Outline

- A Multicycle Implementation
- Mapping Control to Hardware (D.3, D.4)
- Microprogramming: Simplifying Control Design (D.5)
- Concluding Remarks

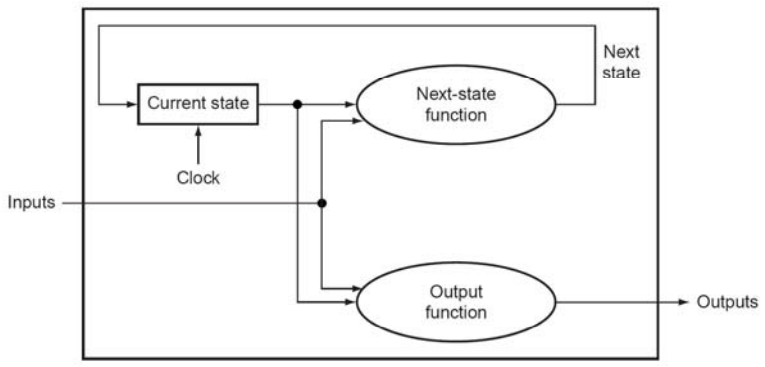
Implementing the Control

- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've accumulated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

Department of Electrical Engineering, Feng-Chia University

Review: Finite State Machines

- Finite State Machines:
  - a set of states and
  - next state function (determined by current state and the input)
  - output function (determined by current state and possibly input)

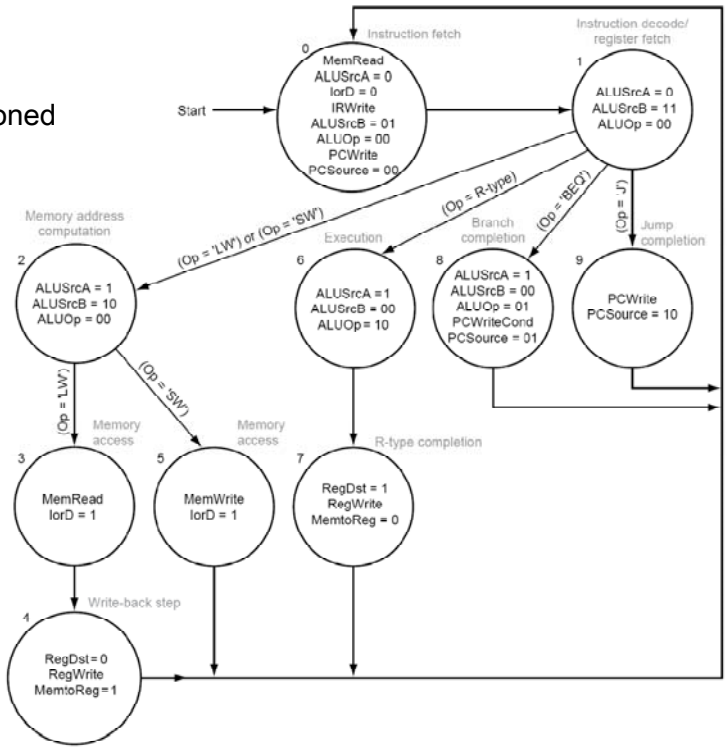


  - We'll use a Moore machine (output based only on current state)

Text Book : D9

Graphical Specification of FSM

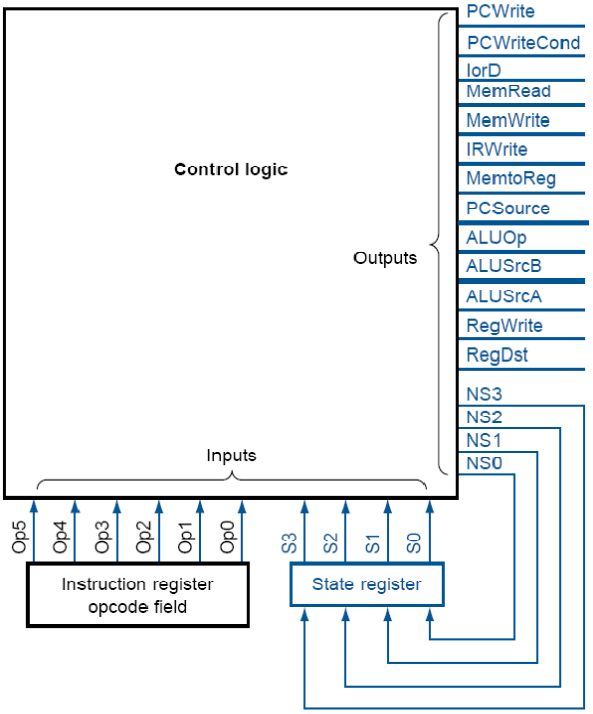
- Note:
  - don't care if not mentioned
  - asserted if name only
  - otherwise exact value
- How many state bits will we need?



Text Book : D10 ~ D11

Finite State Machine for Control

- Implementation:



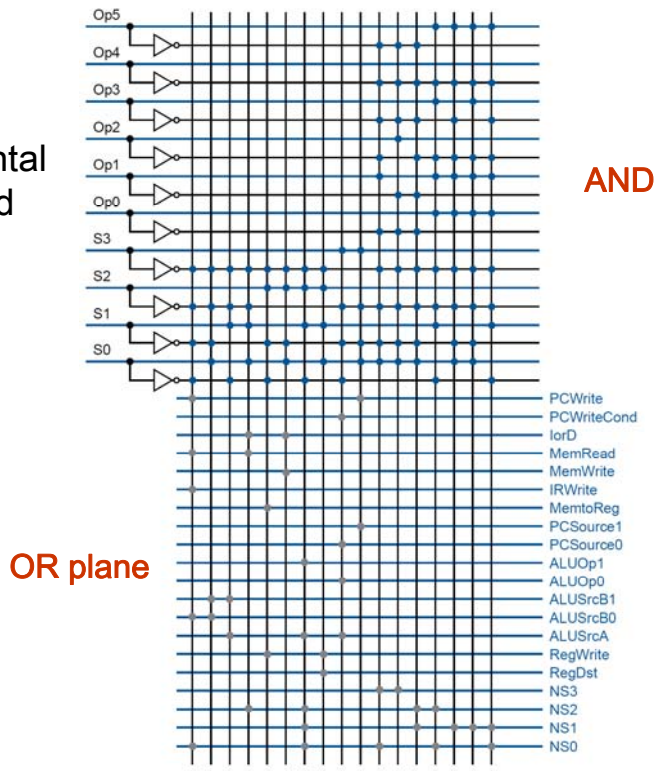
Output	Current states	Op
PCWrite	state0 + state9	
PCWriteCond	state8	
IorD	state3 + state5	
MemRead	state0 + state3	
MemWrite	state5	
IRWrite	state0	
MemtoReg	state4	
PCSource1	state9	
PCSource0	state8	
ALUOp1	state6	
ALUOp0	state8	
ALUSrcB1	state1 +state2	
ALUSrcB0	state0 + state1	
ALUSrcA	state2 + state6 + state8	
RegWrite	state4 + state7	
RegDst	state7	
NextState0	state4 + state5 + state7 + state8 + state9	
NextState1	state0	
NextState2	state1	(Op = 'lw') + (Op = 'sw')
NextState3	state2	(Op = 'lw')
NextState4	state3	
NextState5	state2	(Op = 'sw')
NextState6	state1	(Op = 'R-type')
NextState7	state6	
NextState8	state1	(Op = 'beq')
NextState9	state1	(Op = 'jmp')

Department of Electrical Engineering, Feng-Chia University

Text Book : D20 ~ D21

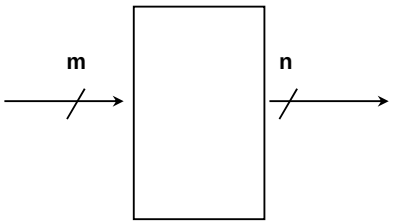
PLA Implementation

- PLA: Programmed Logic Array
- If I picked a horizontal or vertical line could you explain it?



ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM.
  - our outputs are the bits of data that the address points to.



0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	0	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

2^m is the "height", and n is the "width"

Department of Electrical Engineering, Feng-Chia University

ROM Implementation

- How many inputs are there?
  - 6 bits for opcode, 4 bits for state = 10 address lines (i.e., 2^{10} = 1024 different addresses)
- How many outputs are there?
  - 16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is 2^{10} x 20 = 20K bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
  - i.e., opcode is often ignored

ROM vs PLA

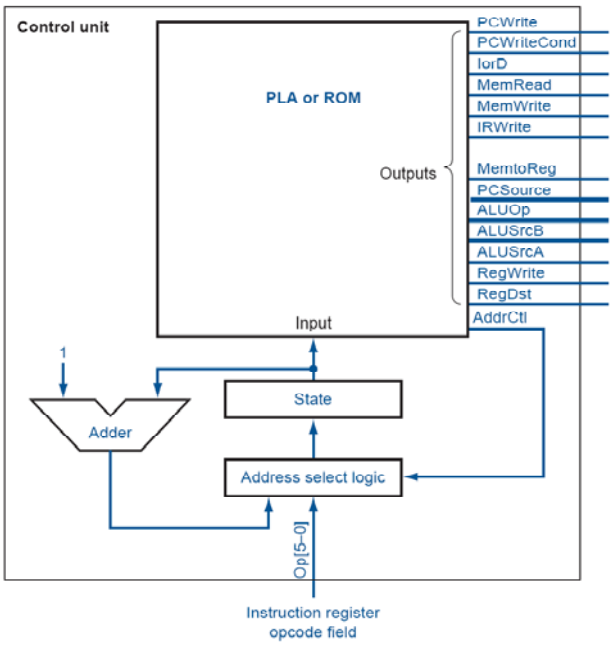
- Break up the table into two parts
  - 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM
  - 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM
  - Total: 4.3K bits of ROM
- PLA is much smaller
  - can share product terms
  - only need entries that produce an active output
  - can take into account don't cares
- Size is (#inputs x #product-terms) + (#outputs x #product-terms)
  For this example = (10x17)+(20x17) = 510 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

Department of Electrical Engineering, Feng-Chia University

Text Book : D22 ~ D23

Another Implementation Style

- Complex instructions: the "next state" is often current state + 1

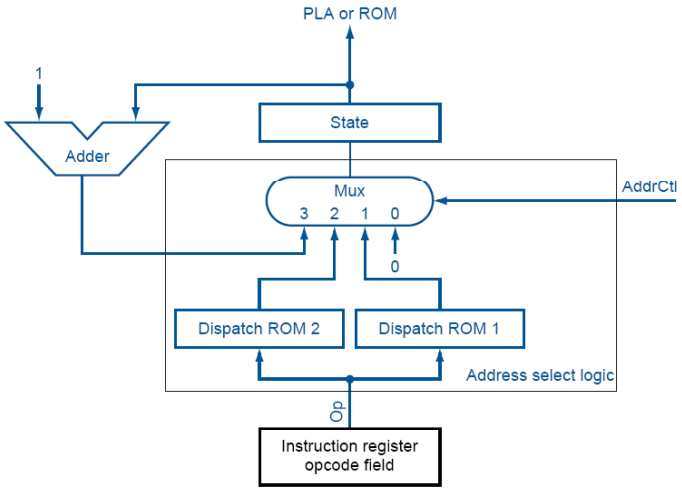


Text Book : D24 ~ D26

Details

Dispatch ROM 1		
Op	**Opcode name**	**Value**
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	**Opcode name**	**Value**
100011	lw	0011
101011	sw	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

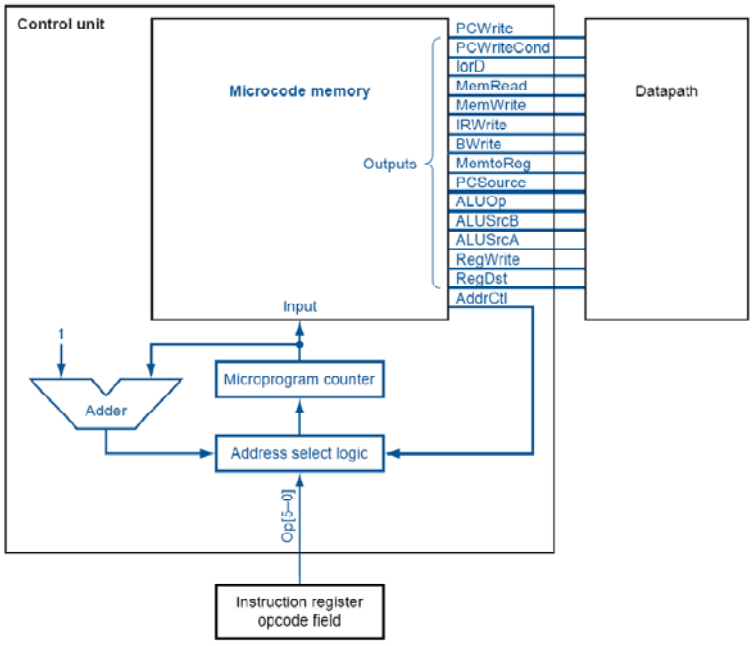
Department of Electrical Engineering, Feng-Chia University

Outline

- A Multicycle Implementation
- Mapping Control to Hardware (D.3, D.4)
- Microprogramming: Simplifying Control Design (D.5)
- Concluding Remarks

Microprogramming

- What are the “microinstructions” ?



Department of Electrical Engineering, Feng-Chia University

Microprogramming

- A specification methodology
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- *Will two implementations of the same architecture have the same microcode?*
- *What would a microassembler do?*

Computer Organization and Architecture, Fall 2010

Microinstruction format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

Department of Electrical Engineering, Feng-Chia University

Maximally vs. Minimally Encoded

- No encoding:
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!
- Lots of encoding:
  - send the microinstructions through logic to get control signals
  - uses less memory, slower
- Historical context of CISC:
  - Too much logic to put on a single chip with everything else
  - Use a ROM (or even RAM) to hold the microcode
  - It's easy to add new instructions

Computer Organization and Architecture, Fall 2010

Microcode: Trade-offs

- Distinction between specification and implementation is sometimes blurred
- Specification Advantages:
  - Easy to design and write
  - Design architecture and microcode in parallel
- Implementation (off-chip ROM) Advantages
  - Easy to change since values are in memory
  - Can emulate other architectures
  - Can make use of internal registers
- Implementation Disadvantages, SLOWER now that:
  - Control is implemented on same chip as processor
  - ROM is no longer faster than RAM
  - No need to go back and make changes

Department of Electrical Engineering, Feng-Chia University

Historical Perspective

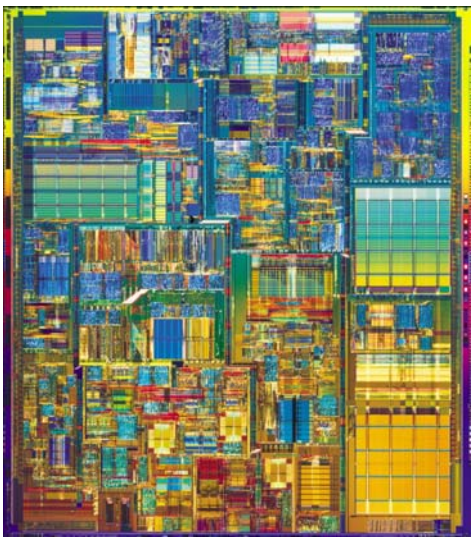
- In the '60s and '70s microprogramming was very important for implementing machines
- This led to more sophisticated ISAs and the VAX
- In the '80s RISC processors based on pipelining became popular
- Pipelining the microinstructions is also possible!
- Implementations of IA-32 architecture processors since 486 use:
  - "hardwired control" for simpler instructions
    (few cycles, FSM control implemented using PLA or random logic)
  - "microcoded control" for more complex instructions
    (large numbers of cycles, central control store)
- The IA-64 architecture uses a RISC-style ISA and can be implemented without a large central control store

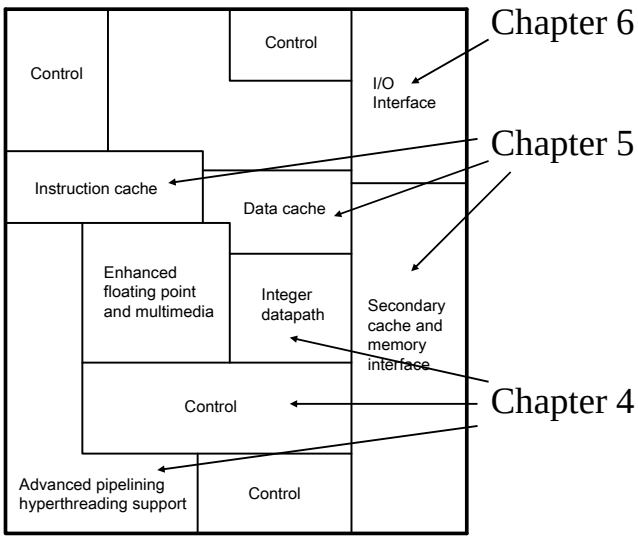
Outline

- A Multicycle Implementation
- Mapping Control to Hardware (D.3, D.4)
- Microprogramming: Simplifying Control Design (D.5)
- Concluding Remarks

Pentium 4

- Pipelining is important (last IA-32 without it was 80386 in 1985)
- Pipelining is used for the simple instructions favored by compilers





Summary

- If we understand the instructions…
  We can build a simple processor !
- If instructions take different amounts of time, multi-cycle is better
- Datapath implemented using:
  - Combinational logic for arithmetic
  - State holding elements to remember bits
- Control implemented using:
  - Combinational logic for single-cycle implementation
  - Finite state machine for multi-cycle implementation

The Big Picture