Computer Architecture and Organization

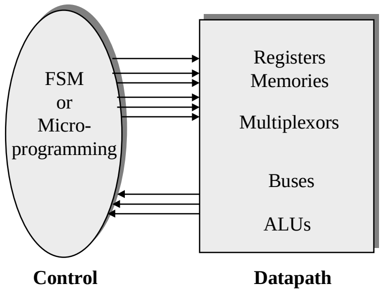
Chapter 5 Processor: Datapath and Control

Henk Corporaal / Ben Juurlink
Delft University of Technology
Spring 2000
Email: h.corporaal@et.tudelft.nl
URL : http://cardit.et.tudelft.nl/~heco/courses/et2-010

Topics

- Building a datapath
  - support a subset of the MIPS-I instruction-set
- A single cycle processor datapath
  - all instruction actions in one (long) cycle
- A multicycle processor datapath
  - each instructions takes multiple (shorter) cycles
- Control: microprogramming
- Exception support
- Real stuff: Pentium Pro/II/III implementation

Datapath and Control



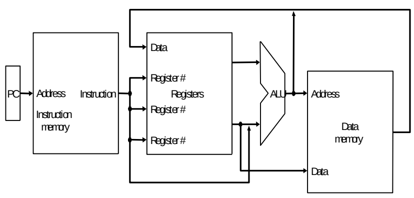
The Processor: Datapath & Control

- We're ready to look at an implementation of the MIPS
- Simplified to contain only:
  - memory-reference instructions: `lw, sw`
  - arithmetic-logical instructions: `add, sub, and, or, slt`
  - control flow instructions: `beq, j`
- Generic Implementation:
  - use the program counter (PC) to supply instruction address
  - get the instruction from memory
  - read registers
  - use the instruction to decide exactly what to do

- All instructions use the ALU after reading the registers
  Why? memory-reference? arithmetic? control flow?

More Implementation Details

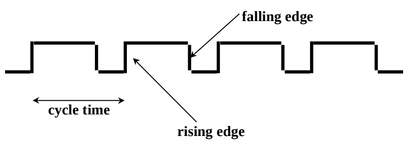
- Abstract / Simplified View:



- Two types of functional units:
  - elements that operate on data values (combinational)
  - elements that contain state (sequential)

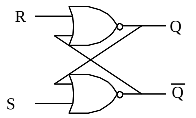
State Elements

- Unclocked vs. Clocked
- Clocks used in synchronous logic
  - when should an element that contains state be updated?



An unclocked state element

- The set-reset (SR) latch
  - output depends on present inputs and also on past inputs



Truth table:

R	S	Q
0	0	Q
0	1	1
1	0	0
1	1	?

state change

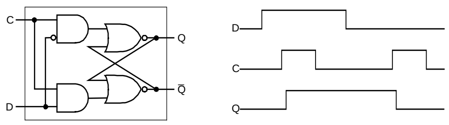
Latches and Flip-flops

- Output is equal to the stored value inside the element (don't need to ask for permission to look at the value)
- Change of state (value) is based on the clock
  - Latches: whenever the inputs change, and the clock is asserted
  - Flip-flop: state changes only on a clock edge (edge-triggered methodology)

A clocking methodology defines when signals can be read and written — wouldn't want to read a signal at the same time it was being written

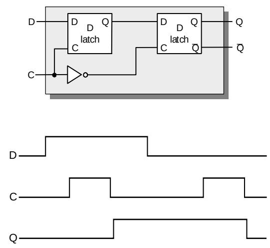
D-latch

- Two inputs:
  - the data value to be stored (D)
  - the clock signal (C) indicating when to read & store D
- Two outputs:
  - the value of the internal state (Q) and it's complement



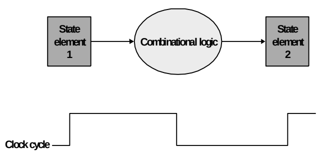
D flip-flop

- Output changes only on the clock edge



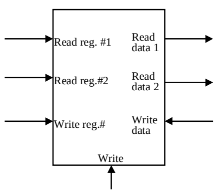
Our Implementation

- An edge triggered methodology
- Typical execution:
  - read contents of some state elements,
  - send values through some combinational logic
  - write results to one or more state elements



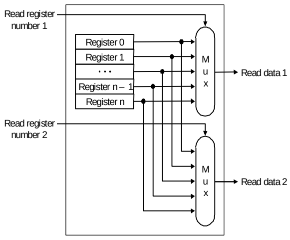
Register File

- 3-ported: one write, two read ports



Register file read ports

- Register file built using D flip-flops

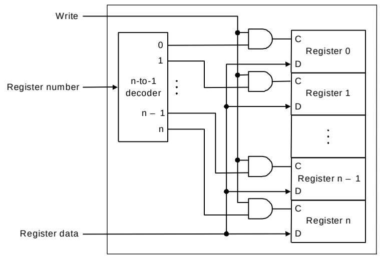


Implementation of the read ports

computer architecture and organization 13

Register file write port

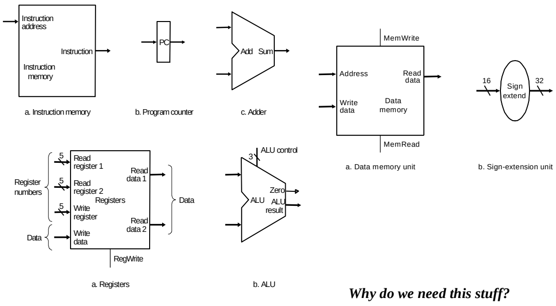
- Note: we still use the real clock to determine when to write



computer architecture and organization 14

Simple Implementation

- Include the functional units we need for each instruction

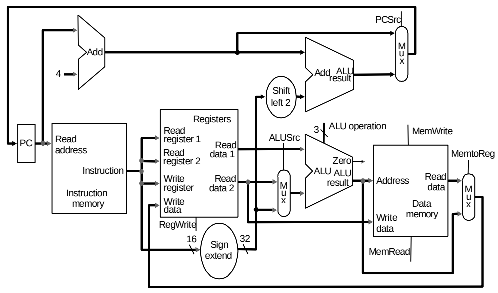


Why do we need this stuff?

computer architecture and organization 15

Building the Datapath

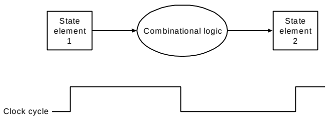
- Use multiplexors to stitch them together



computer architecture and organization 16

Our Simple Control Structure

- All of the logic is combinational
- We wait for everything to settle down, and the right thing to be done
  - ALU might not produce “right answer” right away
  - we use write signals along with clock to determine when to write
- Cycle time determined by length of the longest path



We are ignoring some details like setup and hold times !

computer architecture and organization 17

Control

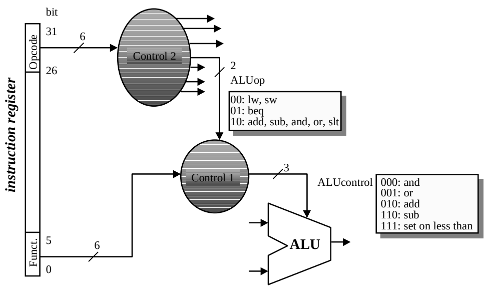
- Selecting the operations to perform (ALU, read/write, etc.)
- Controlling the flow of data (multiplexor inputs)
- Information comes from the 32 bits of the instruction
- Example:
  add $8, $17, $18 Instruction Format:

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- ALU’s operation based on instruction type and function code

computer architecture and organization 18

Control: 2 level implementation



Datapath with Control

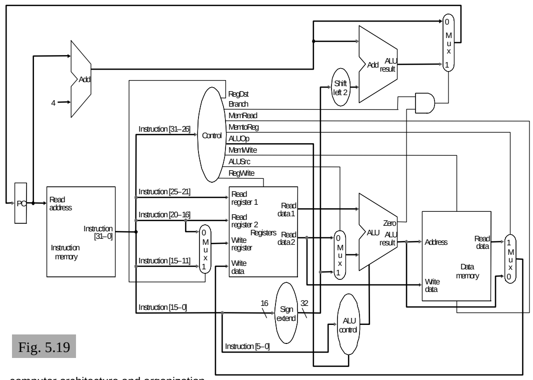


Fig. 5.19

ALU Control1

- What should the ALU do with this instruction example: lw $1, 100($2)

35	2	1	100
op	rs	rt	16 bit offset

- ALU control input

```
000  AND
001  OR
010  add
110  subtract
111  set-on-less-than
```

- Why is the code for subtract 110 and not 011?

ALU Control1

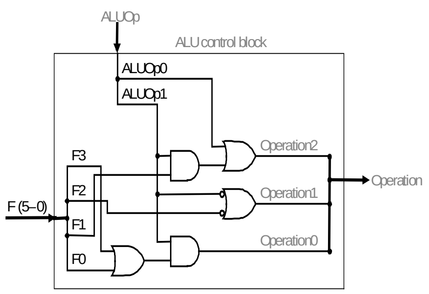
- Must describe hardware to compute 3-bit ALU control input
  - given instruction type
    - 00 = lw, sw
    - 01 = beq,
    - 10 = arithmetic

    ALU Operation computed from instruction type
  - function code for arithmetic
- Describe it using a truth table (can turn into gates):

ALUOp		Funct field						Operation
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
X	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

ALU Control1

- Simple combinational logic (truth tables)



Deriving Control2 signals

Input — 9 control (output) signals

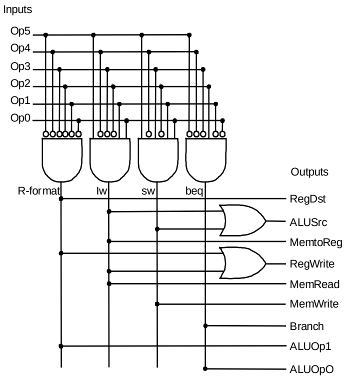
Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

Determine these control signals directly from the opcodes:

R-format:	0
lw:	35
sw:	43
beq:	4

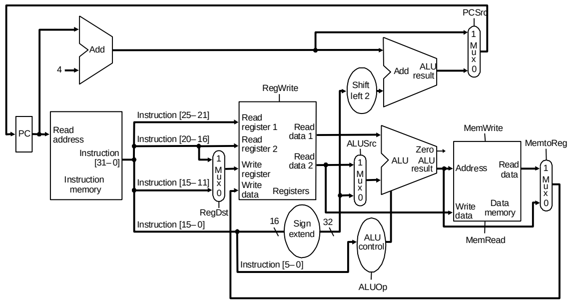
Control 2

- PLA example implementation



Single Cycle Implementation

- Calculate cycle time assuming negligible delays except:
  - memory (2ns), ALU and adders (2ns), register file access (1ns)

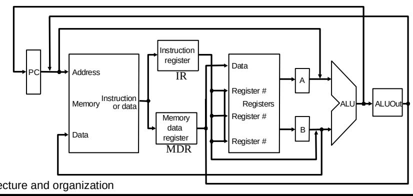


Single Cycle Implementation

- Memory (2ns), ALU & adders (2ns), reg. file access (1ns)
- Fixed length clock: longest instruction is the 'lw' which requires 8 ns
- Variable clock length (not realistic, just as exercise):
  - R-instr: 6 ns
  - Load: 8 ns
  - Store: 7 ns
  - Branch: 5 ns
  - Jump: 2 ns
- Average depends on instruction mix (see pg 374)

Where we are headed

- Single Cycle Problems:
  - what if we had a more complicated instruction like floating point?
  - wasteful of area: *NO Sharing of Hardware resources*
- One Solution:
  - use a "smaller" cycle time
  - have different instructions take different numbers of cycles
  - a "multicycle" datapath:

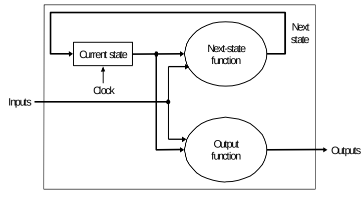


Multicycle Approach

- We will be reusing functional units
  - ALU used to compute address and to increment PC
  - Memory used for instruction and data
- Add registers after every major functional unit
- Our control signals will not be determined solely by instruction
  - e.g., what should the ALU do for a "subtract" instruction?
- We'll use a finite state machine (FSM) or microcode for control

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)

Review: finite state machines

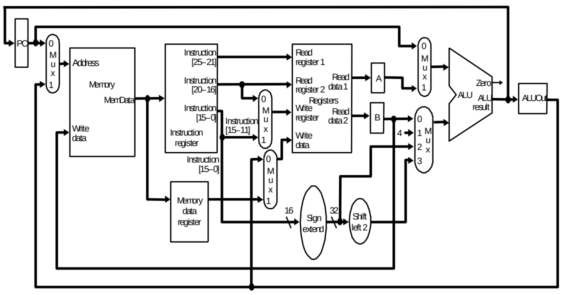
- Example:

B. 21 *A friend would like you to build an "electronic eye" for use as a fake security device. The device consists of three lights lined up in a row, controlled by the outputs Left, Middle, and Right, which, if asserted, indicate that a light should be on. Only one light is on at a time, and the light "moves" from left to right and then from right to left, thus scaring away thieves who believe that the device is monitoring their activity. Draw the graphical representation for the finite state machine used to specify the electronic eye. Note that the rate of the eye's movement will be controlled by the clock speed (which should not be too great) and that there are essentially no inputs.*

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
- Notice: we distinguish
  - processor state: programmer visible registers
  - internal state: programmer invisible registers (like IR, MDR, A, B, and ALUout)

Multicycle Approach



Multicycle Approach

- Note that previous picture does not include:
  - branch support
  - jump support
  - Control lines and logic
- For complete picture see fig 5.33 page 383
- Tclock > max (ALU delay, Memory access, Regfile access)

Five Execution Steps

- Instruction Fetch
- Instruction Decode and Register Fetch
- Execution, Memory Address Computation, or Branch Completion
- Memory Access or R-type instruction completion
- 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?

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
- Previous two actions are done optimistically!!
- 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 (instruction dependent)

- ALU is performing one of four 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;`
- Jump:
  `PC = PC[31-28] || (IR[25-0]<<2)`

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

Write-back step

- Memory read completion step

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

What about all the other instructions?

Summary execution steps

Steps taken to execute any instruction class

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) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
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?

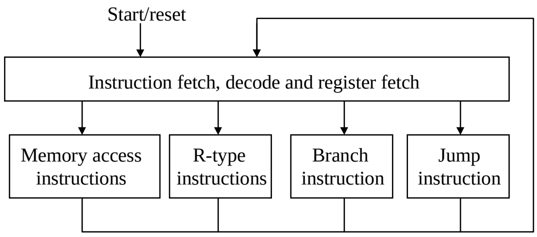
```
    lw $t2, 0($t3)
    lw $t3, 4($t3)
    beq $t2, $t3, L1     #assume not taken
    add $t5, $t2, $t3
    sw $t5, 8($t3)
L1:...
```

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

Implementing the Control

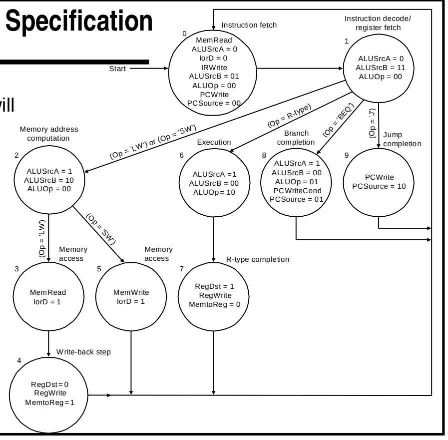
- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we have accumulated to specify a finite state machine (FSM)
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be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

FSM: high level view



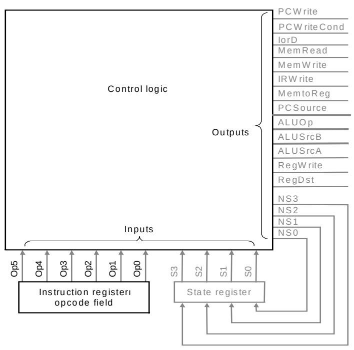
Graphical Specification of FSM

- How many state bits will we need?



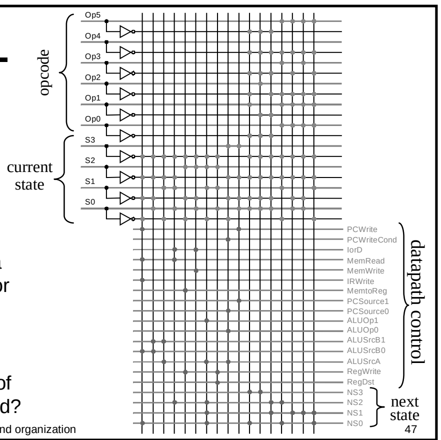
Finite State Machine for Control

- Implementation:



PLA Implementation

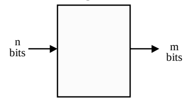
(see fig C.14)



- If I picked a horizontal or vertical line could you explain it ?
- What type of FSM is used?

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



address	data
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

m is the "heigth", and n is the "width"

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
  (very large 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 Implementation

Cheaper implementation:

- Exploit the fact that the FSM is a Moore machine ==>
  - Control outputs only depend on current state and not on other incoming control signals !
  - Next state depends on all inputs
- 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 number of bits: 4.3K bits of ROM

ROM vs PLA

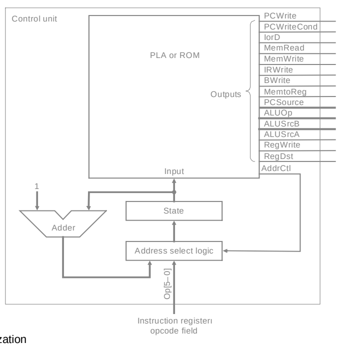
- PLA is much smaller
  - can share product terms (ROM has an entry (=address) for every product term
  - only need entries that produce an active output
  - can take into account don't cares
- Size of PLA:
  (#inputs $\times$ #product-terms) + (#outputs $\times$ #product-terms)
  - For this example: (10x17)+(20x17) = 460 PLA cells
- PLA cells usually slightly bigger than the size of a ROM cell

Another Implementation Style

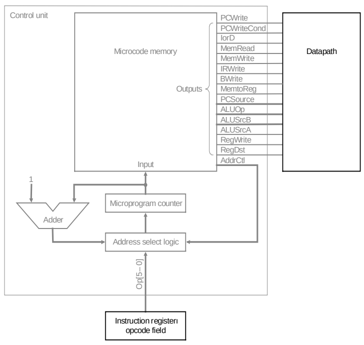
- Real machines have many instructions => complex FSM with many states
  - Graphical specification becomes cumbersome
- Specify control as an instruction
  - microinstructions
  - built out of separate fields (for controlling ALU, SRC1, SCR2, etc)
- Exploit the fact that usually the next state is the next microinstruction (just like in a sequential programming language)
  - default sequencing
  - use micro program counter (indicating next state = next instr.)

Another Implementation Style

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



Micro-programming



What are the “microinstructions” ?

Microinstruction format

- Each microinstruction contains 7 fields

Field name	bits	Function of field
ALU control	2	Specify ALU operation
SRC1	1	source for first ALU operand
SRC2	2	source for second ALU op.
Register control	2	read/write reg.file & source of write value
Memory	2	read/write mem. & mem. source
PCWrite control	2	writing PC with ALU ouput (cond.) or Jump addres
Sequencing	2	choose next instr: Seq/Fetch/Dispatch to ROM1 or ROM 2

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, lorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, lorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, lorD = 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.

Microprogramming

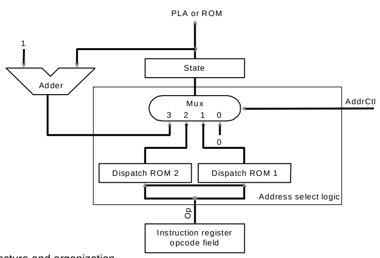
- A specification methodology
  - appropriate if hundreds of opcodes, modes, cycles, etc.
  - signals specified symbolically using microinstructions

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

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



Details

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

Microprogramming

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

Maximally vs. Minimally Encoded

- No encoding (also called horizontal encoding, or 1-hot encoding):
  - 1 bit for each datapath operation
  - faster, requires more memory (logic)
  - used for Vax 780 — an astonishing 400K of memory!
- Lots of encoding (also called vertical 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

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

Exceptions

- Unexpected events
- External: interrupt
  - e.g. I/O request
- Internal: exception
  - e.g. Overflow, Undefined instruction opcode, Software trap
- How to handle exception?
  - Jump to general entry point (record exception type in status register)
  - Jump to vectored entry point
  - Address of faulting instruction has to be recorded !

Exceptions

Changes needed: see fig. 5.48 / 5.49 / 5.50

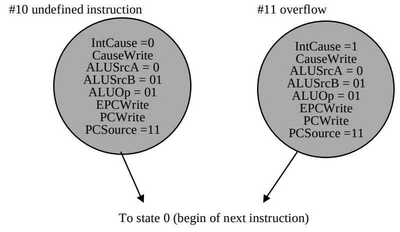
- extend PC input mux with extra entry with fixed address: "C000000hex"
- Add EPC register containing old PC (we'll use the ALU to decrement PC with 4)
  - extra input ALU src2 needed with fixed value 4
- Cause register (one bit in our case) containing:
  - 0: undefined instruction
  - 1: ALU overflow
- Add 2 states to FSM
  - undefined instr. state #10
  - overflow state #11

Exceptions

Legend:

IntCause =0/1	type of exception
CauseWrite	write Cause register
ALUSrcA = 0	select PC
ALUSrcB = 01	select constant 4
ALUOp = 01	subtract operation
EPCWrite	write EPC register with current PC
PCWrite	write PC with exception address
PCSource =11	select exception address: C000000hex

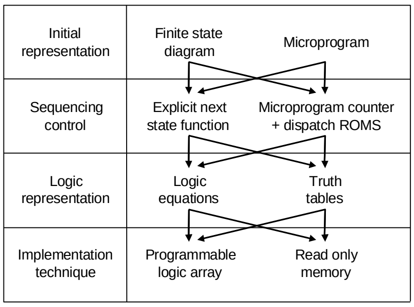
2 New states:



Pentium Pro / II

- Use multicycle data path for 80x86 instructions
- Combine hardwired (FSM) control for simple instructions with microcoded control for complex instructions (since 80486)
- Pentium Pro:
  - internal RISC engine executing micro-operations (of 72 bit)
  - multiple FUs
  - up to four 80x86 instructions issued per cycle and translated into micro-operations (by set of PLAs generating 1200 different micro-operations)
  - complex 80x86 instructions are handled by micro-code (8000 micro-instructions)
  - four micro-operations issued per cycle (4x72 bits expand into 120 Int and 285 FP control lines)

The Big Picture



computer architecture and organization 67

Exercises

From Chapter five:

- 5.1, 5.3
- 5.5, 5.6
- 5.9
- 5.12

computer architecture and organization 68