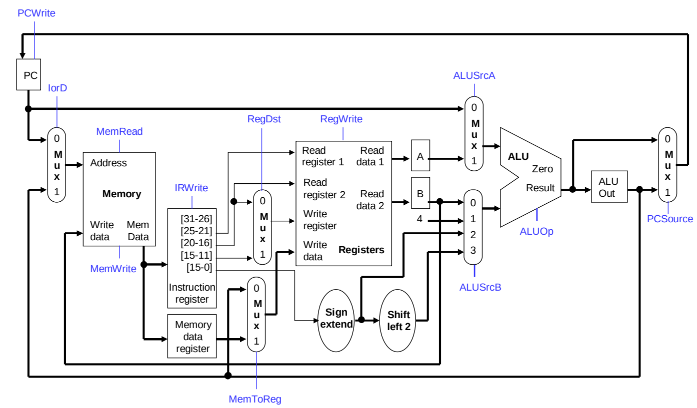
Lecture 10 (Wed 10/15/2008)

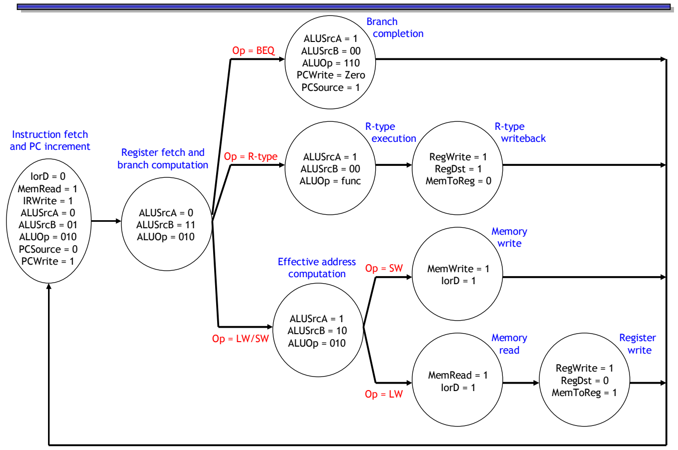
- Lab #1 Hardware – Due Fri Oct 17
- HW #2 – MIPS programming, due Wed Oct 22
- Midterm – Fri Oct 24

- Today's objectives:
  – Microprogramming
  – Extending the multi-cycle datapath
  – Multi-cycle performance

The multicycle datapath



Finite-state machine for the control unit



Implementing the FSM

- This can be translated into a state table; here are the first two states.

Current State	Input (Op)	Next State	Output (Control signals)											
			PC Write	IorD	Mem Read	Mem Write	IR Write	Reg Dst	MemTo Reg	Reg Write	ALU SrcA	ALU SrcB	ALU Op	PC Source
Instr Fetch	X	Reg Fetch	1	0	1	0	1	X	X	0	0	01	010	0
Reg Fetch	BEQ	Branch compl	0	X	0	0	0	X	X	0	0	11	010	X
Reg Fetch	R-type	R-type execute	0	X	0	0	0	X	X	0	0	11	010	X
Reg Fetch	LW/SW	Compute eff addr	0	X	0	0	0	X	X	0	0	11	010	X

- You can implement this the hard way.
  – Represent the current state using flip-flops or a register.
  – Find equations for the next state and (control signal) outputs in terms of the current state and input (instruction word).
- Or you can use the easy way.
  – Stick the whole state table into a memory, like a ROM.
  – This would be much easier, since you don't have to derive equations.

Pitfalls of state machines

- As mentioned last time, we could translate this state diagram into a state table, and then make a logic circuit or stick it into a ROM.
- This works pretty well for our small example, but designing a finite-state machine for a larger instruction set is much harder.
  - There could be many states in the machine. For example, some MIPS instructions need 20 stages to execute in some implementations—each of which would be represented by a separate state.
  - There could be many paths in the machine. For example, the DEC VAX from 1978 had nearly 300 opcodes... that's a lot of branching!
  - There could be many outputs. For instance, the Pentium Pro's integer datapath has 120 control signals, and the floating-point datapath has 285 control signals.
  - Implementing and maintaining the control unit for processors like these would be a nightmare. You'd have to work with large Boolean equations or a huge state table.

Motivation for microprogramming

- Think of the control unit's state diagram as a little program.
  - Each state represents a "command," or a set of control signals that tells the datapath what to do.
  - Several commands are executed sequentially.
  - "Branches" may be taken depending on the instruction opcode.
  - The state machine "loops" by returning to the initial state.
- Why don't we invent a special language for making the control unit?
  - We could devise a more readable, higher-level notation rather than dealing directly with binary control signals and state transitions.
  - We would design control units by writing "programs" in this language.
  - We will depend on a hardware or software translator to convert our programs into a circuit for the control unit.

A good notation is very useful

- Instead of specifying the exact binary values for each control signal, we will define a symbolic notation that's easier to work with.
- As a simple example, we might replace ALUSrcB = 01 with ALUSrcB = 4.
- We can also create symbols that *combine* several control signals together. Instead of

  IorD = 0
  MemRead = 1
  IRWrite = 1

  it would be nicer to just say something like

  Read PC

Microinstructions

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next

- For the MIPS multicycle we could define microinstructions with eight fields.
  - These fields will be filled in symbolically, instead of in binary.
  - They determine all the control signals for the datapath. There are only 8 fields because some of them specify more than one of the 12 actual control signals.
  - A microinstruction corresponds to one execution stage, or one cycle.
- You can see that in each microinstruction, we can do something with the ALU, register file, memory, and program counter units.

Specifying ALU operations

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next

- ALU control selects the ALU operation.
  - Add indicates addition for memory offsets or PC increments.
  - Sub performs source register comparisons for “beq”.
  - Func denotes the execution of R-type instructions.
- SRC1 is either PC or A, for the ALU’s first operand.
- SRC2, the second ALU operand, can be one of four different values.
  - B for R-type instructions and branch comparisons.
  - The constant 4 to increment the PC.
  - Extend, the sign-extended constant field for _______________.
  - Extshift, the sign-extended, shifted constant _______________.
- These correspond to the ALUOp, ALUSrcA and ALUSrcB control signals, except we use names like “Add” and not actual bits like “010.”

Specifying register and memory actions

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next

- Register control selects a register file action.
  - Read to read from registers “rs” and “rt” of the instruction word.
  - Write ALU writes ALUOut into destination register “rd”.
  - Write MDR saves MDR into destination register “rt”.
- Memory chooses the memory unit’s action.
  - Read PC reads an instruction from address PC into IR.
  - Read ALU reads data from address ALUOut into MDR.
  - Write ALU writes register B to address memory ALUOut.

Specifying PC actions

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next

- PCWrite control determines what happens to the PC.
  - ALU sets PC to ALUOut, used in incrementing the PC.
  - ALU-Zero writes ALUOut to PC only if the ALU’s Zero condition is true. This is used to complete a branch instruction.
- Next determines the next microinstruction to be executed.
  - Seq causes the next microinstruction to be executed.
  - Fetch returns to the initial instruction fetch stage.
  - Dispatch i is similar to a “switch” or “case” statement; it branches depending on the actual instruction word.

The first stage, the microprogramming way

- Below are two lines of microcode to implement the first two multicycle execution stages, instruction fetch and register fetch.
- The first line, labelled Fetch, involves several actions.
  - Read from memory address PC.
  - Use the ALU to compute PC + 4, and store it back in the PC.
  - Continue on to the next sequential microinstruction.

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshift	Read			Dispatch 1

The second stage

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshift	Read			Dispatch 1

- The second line implements the register fetch stage.
  - Read registers rs and rt from the register file.
  - Pre-compute PC + (sign-extend(IR[15-0]) << 2) for branches.
  - Determine the next microinstruction based on the opcode of the current MIPS program instruction.

```
switch (opcode) {
    case 4:   goto BEQ1;
    case 0:   goto Rtype1;
    case 43:
    case 35:  goto Mem1;
}
```

13

Completing a beq instruction

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
BEQ1							

- Control would transfer to this microinstruction if the opcode was "beq".
  - Compute A-B, to set the ALU's Zero bit if A=B.
  - Update PC with ALUOut (which contains the branch target from the previous cycle) if Zero is set.
  - The beq is completed, so fetch the next instruction.
- The 1 in the label BEQ1 reminds us that we came here via the first branch point ("dispatch table 1"), from the second execution stage.

14

Completing an arithmetic instruction

Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
Rtype1							

- What if the opcode indicates an R-type instruction?
  - The first cycle here performs an operation on registers A and B, based on the MIPS instruction's func field.
  - The next stage writes the ALU output to register "rd" from the MIPS instruction word.
- We can then go back to the Fetch microinstruction, to fetch and execute the next MIPS instruction.

15

Completing data transfer instructions

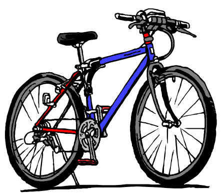
Label	ALU control	Src1	Src2	Register control	Memory	PCWrite control	Next
Mem1							Dispatch 2
SW2							
LW2							

- For both sw and lw instructions, we should first compute the effective memory address, A + sign-extend(IR[15-0]).
- Another dispatch or branch distinguishes between stores and loads.
  - For sw, we store data (from B) to the effective memory address.
  - For lw we copy data from the effective memory address to register rt.
- In either case, we continue on to Fetch after we're done.

16

Microprogramming vs. programming

- Microinstructions correspond to control signals.
  - They describe what is done in a single clock cycle.
  - These are the most basic operations available in a processor.
- Microprograms implement higher-level MIPS instructions.
  - MIPS assembly language instructions are comparatively complex, each possibly requiring multiple clock cycles to execute.
  - But each complex MIPS instruction can be implemented with several simpler microinstructions.



Similarities with assembly language

- Microcode is intended to make control unit design easier.
  - We defined symbols like Read PC to replace binary control signals.
  - A translator can convert microinstructions into a real control unit.
  - The translation is straightforward, because each microinstruction corresponds to one set of control values.
- This sounds similar to MIPS assembly language!
  - We use mnemonics like lw instead of binary opcodes like 100011.
  - MIPS programs must be assembled to produce real machine code.
  - Each MIPS instruction corresponds to a 32-bit instruction word.

Managing complexity

- It looks like all we've done is devise a new notation that makes it easier to specify control signals.
- That's exactly right! It's all about managing complexity.
  - Control units are probably the most challenging part of CPU design.
  - Large instruction sets require large state machines with many states, branches and outputs.
  - Control units for multicycle processors are difficult to create and maintain.
- Applying programming ideas to hardware design is a useful technique.

Situations when microprogramming is bad

- One disadvantage of microprograms is that looking up control signals in a ROM can be *slower* than generating them from simplified circuits.
- Sometimes complex instructions implemented in hardware are *slower* than equivalent assembly programs written using simpler instructions
  - Complex instructions are usually very general, so they can be used more often. But this also means they can't be optimized for specific operands or situations.
  - Some microprograms just aren't written very efficiently. But since they're built into the CPU, people are stuck with them (at least until the next processor upgrade).

How microcode is used today

- Modern CISC processors (like x86) use a combination of hardwired logic and microcode to balance design effort with performance.
  - Control for many simple instructions can be implemented in hardwired logic
  - Less-used or very complex instructions are microprogrammed to make the design easier and more flexible.
- In this way, designers observe the "first law of performance"
  - Make the common case fast!

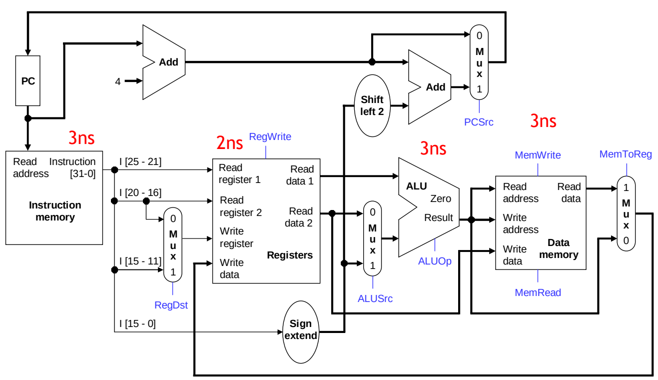
21

DEC VAX 11/780

- The VAX was designed in 1978 by Digital Equipment Corporation.
- It has one of the most complex instruction sets ever. (Compiler technology wasn't very good back then, and they wanted to make assembly programming easier.)
- VMS, the VAX multiuser, cluster-based operating system, was designed by Dave Cutler, who was also in charge of Windows NT.
- The VAX had a 32-bit processor, seven years before Intel's 80386.
- The cycle time was 200ns. 5MHz!
- All of this cost $200,000.

22

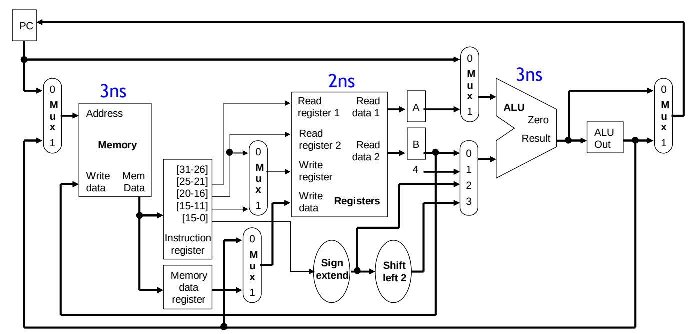
The single-cycle datapath; what is the cycle time?



23

Performance of a multicycle implementation

- Let's assume the following delays for the major functional units.



24

Comparing cycle times

- The clock period has to be long enough to allow all of the required work to complete within the cycle.
- In the single-cycle datapath, the "required work" was just the complete execution of any instruction.
  - The longest instruction, lw, requires 13ns (3 + 2 + 3 + 3 + 2).
  - So the clock cycle time has to be 13ns, for a 77MHz clock rate.
- For the multicycle datapath, the "required work" is only a single stage.
  - The longest delay is 3ns, for both the ALU and the memory.
  - So our cycle time has to be 3ns, or a clock rate of 333MHz.
  - The register file needs only 2ns, but it must wait an extra 1ns to stay synchronized with the other functional units.
- The single-cycle cycle time is limited by the slowest *instruction*, whereas the multicycle cycle time is limited by the slowest *functional unit*.

Comparing instruction execution times

- In the single-cycle datapath, each instruction needs an entire clock cycle, or 13ns, to execute.
- With the multicycle CPU, different instructions need different numbers of clock cycles, and hence different amounts of time.
  - A branch needs 3 cycles, or 3 x 3ns = 9ns.
  - Arithmetic and sw instructions each require 4 cycles, or 12ns.
  - Finally, a lw takes 5 stages, or 15ns.
- We can make some observations about performance already.
  - Loads take *longer* with this multicycle implementation, while all other instructions are faster than before.
  - So if our program doesn't have too many loads, then we should see an increase in performance.

The gcc example

- Let's assume the gcc instruction mix:

Instruction	Frequency
Arithmetic	48%
Loads	22%
Stores	11%
Branches	19%

- In a single-cycle datapath, all instructions take 13ns to execute.
- The average execution time for an instruction on the multicycle processor works out to 12.09ns.

(48% x 12ns) + (22% x 15ns) + (11% x 12ns) + (19% x 9ns) = 12.09ns

- The multicycle implementation is faster in this case, but not by much. The speedup here is only 7.5%.

13ns / 12.09ns = 1.075

This CPU is too simple

- Our example instruction set is too simple to see large gains.
  - All of our instructions need about the same number of cycles (3-5).
  - The benefits would be much greater in a more complex CPU, where some instructions require many more stages than others.
- For example, the 80x86 has instructions to push all the registers onto the stack in one shot (PUSHA).
  - Pushing proceeds sequentially, register by register.
  - Implementing this in a single-cycle datapath would be foolish, since the instruction would need a large amount of time to store each register into memory.
  - But the 8086 and VAX are multicycle processors, so these complex instructions don't slow down the cycle time or other instructions.
- Also, recall the real discrepancy between memory speed and processor frequencies.

Multicycle Wrap-up

- A multicycle processor splits instruction execution into several stages, each of which requires one clock cycle.
  - Each instruction can be executed in as few stages as necessary.
- Multicycle control is more complex than the single cycle implementation
  - Extra multiplexers and temporary registers are needed.
  - The control unit must generate *sequences* of control signals.
  - Microprogramming helps manage the complexity by aggregating control signals into groups and using symbolic names
    - Just like assembly is easier than machine code

- Next time, we begin our foray into pipelining.
  - The multicycle implementation makes a good launch point.

29