A x64 processor is screaming along at billions of cycles per second to run the XNU kernel, which is frantically working through all the POSIX-specified abstraction to create the Darwin system underlying OS X, which in turn is straining itself to run Firefox and its Gecko renderer, which creates a Flash object which renders dozens of video frames every second because I wanted to see a cat jump into a box and fall over.
In Review: A Single Cycle Datapath

- We have everything! Now we just need to know how to BUILD CONTROL

```
<table>
<thead>
<tr>
<th>Instruction&lt;31:0&gt;</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
<th>Imm16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- ALUctrl
- MemtoReg
- MemWr
- zero

```

```
<table>
<thead>
<tr>
<th>RegFile</th>
<th>Rs</th>
<th>Rt</th>
<th>Rd</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

- RegDst
- RegWr

```

```
<table>
<thead>
<tr>
<th>Extender</th>
<th>Imm16</th>
<th>16</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- ExtOp
- ALUSrc

```

- Data Memory
- WrEn
- Adr

```
<table>
<thead>
<tr>
<th>Data In</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

- ALU
- clk
- busA
- busB

```

```
<table>
<thead>
<tr>
<th>nPC_sel</th>
<th>instr fetch unit</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

```

- clk
- busW
- 32

```

- cal
- Johnson, Summer 2010 © UCB
5 steps to design a processor

1. Analyze instruction set → datapath requirements
2. Select set of datapath components & establish clock methodology
3. Assemble datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
5. Assemble the control logic
   - Formulate Logic Equations
   - Design Circuits
Step 4: Given Datapath: RTL $\rightarrow$ Control

**Diagram:**
- **Instruction**: $<31:0>$
- **Memory**: Adr, Inst
- **Control**: nPC_sel, RegWr, RegDst, ExtOp, ALUSrc, ALUctr, MemWr, MemtoReg
- **Op** $<2:6>$, **Fun** $<0:5>$, **Rt** $<21:25>$, **Rs** $<16:20>$, **Rd** $<11:15>$, **Imm16** $<0:15>$
- **Data Path**

*Image of diagram with labeled nodes and connections.*
A Summary of the Control Signals (1/2)

<table>
<thead>
<tr>
<th>inst</th>
<th>Register Transfer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>add</strong></td>
<td>R[rd] ← R[rs] + R[rt];  PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = RegB, ALUctr = “ADD”, RegDst = rd, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td><strong>sub</strong></td>
<td>R[rd] ← R[rs] – R[rt];  PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = RegB, ALUctr = “SUB”, RegDst = rd, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td><strong>ori</strong></td>
<td>R[rt] ← R[rs] + zero_ext(Imm16);  PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, Extop = “Z”, ALUctr = “OR”, RegDst = rt, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td><strong>lw</strong></td>
<td>R[rt] ← MEM[ R[rs] + sign_ext(Imm16)];  PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, Extop = “sn”, ALUctr = “ADD”, MemtoReg, RegDst = rt, RegWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td><strong>sw</strong></td>
<td>MEM[ R[rs] + sign_ext(Imm16)] ← R[rs];  PC ← PC + 4</td>
</tr>
<tr>
<td></td>
<td>ALUsrc = Im, Extop = “sn”, ALUctr = “ADD”, MemWr, nPC_sel = “+4”</td>
</tr>
<tr>
<td><strong>beq</strong></td>
<td>if ( R[rs] == R[rt] ) then PC ← PC + sign_ext(Imm16)</td>
</tr>
<tr>
<td></td>
<td>nPC_sel = “br”, ALUctr = “SUB”</td>
</tr>
</tbody>
</table>
## A Summary of the Control Signals (2/2)

### Table

<table>
<thead>
<tr>
<th></th>
<th>add</th>
<th>sub</th>
<th>ori</th>
<th>lw</th>
<th>sw</th>
<th>beq</th>
<th>jump</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>x</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>x</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>nPCsel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>?</td>
</tr>
<tr>
<td>Jump</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x</td>
<td>x</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>x</td>
<td>x</td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>Add</td>
<td>Subtract</td>
<td>Or</td>
<td>Add</td>
<td>Add</td>
<td>Subtract</td>
<td>x</td>
</tr>
</tbody>
</table>

### Control Signals

- **RegDst**: Determines the destination register for the operation.
- **ALUSrc**: Determines the source for the ALU operation.
- **MemtoReg**: Indicates if the memory result should be stored in a register.
- **RegWrite**: Indicates if the result should be written to the destination register.
- **MemWrite**: Indicates if the result should be written to memory.
- **nPCsel**: Selects between the program counter and the branch target for jump instructions.
- **Jump**: Indicates if a jump instruction is being executed.
- **ExtOp**: Extends the operation for longer instructions.
- **ALUctr<2:0>**: Selects between addition, subtraction, or other operations.

### Instruction Types

- **R-type**: op, rs, rt, rd, shamt, funct (add, sub)
- **I-type**: op, rs, rt, immediate (ori, lw, sw, beq)
- **J-type**: op, target address (jump)

### Appendix A

See Appendix A for more detailed information.
Boolean Expressions for Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq (assume ALUctr is 00 ADD, 01: SUB, 10: OR)
ALUctr[1] = or

where,

rtype = ~op_5 • ~op_4 • ~op_3 • ~op_2 • ~op_1 • ~op_0
ori = ~op_5 • ~op_4 • op_3 • op_2 • ~op_1 • op_0
lw = op_5 • ~op_4 • ~op_3 • ~op_2 • op_1 • op_0
sw = op_5 • ~op_4 • op_3 • ~op_2 • op_1 • op_0
beq = ~op_5 • ~op_4 • ~op_3 • op_2 • ~op_1 • ~op_0
jump = ~op_5 • ~op_4 • ~op_3 • ~op_2 • op_1 • ~op_0

add = rtype • func_5 • ~func_4 • ~func_3 • ~func_2 • ~func_1 • ~func_0
sub = rtype • func_5 • ~func_4 • ~func_3 • ~func_2 • func_1 • ~func_0

How do we implement this in gates?
Controller Implementation

```
opcode  func

AND logic
  add
  sub
  ori
  lw
  sw
  beq
  jump

OR logic
  RegDst
  ALUSrc
  MemtoReg
  RegWrite
  MemWrite
  nPCsel
  Jump
  ExtOp
  ALUctr[0]
  ALUctr[1]
```

"AND" logic

"OR" logic
1) MemToReg=‘x’ & ALUctr=‘sub’. SUB or BEQ?

2) ALUctr=‘add’. Which 1 signal is different for all 3 of: ADD, LW, & SW? RegDst or ExtOp?
5 steps to design a processor

1. Analyze instruction set → datapath requirements
2. Select set of datapath components & establish clock methodology
3. **Assemble** datapath meeting the requirements
4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
5. **Assemble** the control logic
   - Formulate Logic Equations
   - Design Circuits
Review: Single cycle datapath

• 5 steps to design a processor
  1. Analyze instruction set datapath requirements
  2. Select set of datapath components & establish clock methodology
  3. Assemble datapath meeting the requirements
  4. Analyze implementation of each instruction to determine setting of control points that effects the register transfer.
  5. Assemble the control logic

• Control is the hard part

• MIPS makes that easier
  • Instructions same size
  • Source registers always in same place
  • Immediates same size, location
  • Operations always on registers/immediates
How We Build The Controller

RegDst = add + sub
ALUSrc = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nPCsel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq (assume ALUctr is 0 ADD, 01: SUB, 10: OR)
ALUctr[1] = or

where,

rtype = ~op5 • ~op4 • ~op3 • ~op2 • ~op1 • ~op0,
ori = ~op5 • ~op4 • op3 • op2 • ~op1 • op0
lw = op5 • ~op4 • ~op3 • ~op2 • op1 • op0
sw = op5 • ~op4 • ~op3 • ~op2 • op1 • op0
beq = ~op5 • ~op4 • ~op3 • op2 • ~op1 • ~op0
jump = ~op5 • ~op4 • ~op3 • ~op2 • op1 • ~op0

add = rtype • func5 • ~func4 • ~func3 • ~func2 • ~func1 • ~func0
sub = rtype • func5 • ~func4 • ~func3 • ~func2 • func1 • ~func0

Omigosh omigosh, do you know what this means?
Processor Performance

• Can we estimate the clock rate (frequency) of our single-cycle processor? We know:
  • 1 cycle per instruction
  • `lw` is the most demanding instruction.
  • Assume these delays for major pieces of the datapath:
    ▪ Instr. Mem, ALU, Data Mem: 2ns each, regfile 1ns
    ▪ Instruction execution requires: 2 + 1 + 2 + 2 + 1 = 8ns
    ▪ ⇒ 125 MHz

• What can we do to improve clock rate?

• Will this improve performance as well?
  • We want increases in clock rate to result in programs executing quicker.
Gotta Do Laundry

- Ann, Brian, Cathy, Dave each have one load of clothes to wash, dry, fold, and put away
  - Washer takes 30 minutes
  - Dryer takes 30 minutes
  - “Folder” takes 30 minutes
  - “Stasher” takes 30 minutes to put clothes into drawers
Sequential Laundry

- Sequential laundry takes 8 hours for 4 loads.
Pipelined Laundry

- Pipelined laundry takes 3.5 hours for 4 loads!
General Definitions

• **Latency**: time to completely execute a certain task
  - for example, time to read a sector from disk is disk access time or disk latency

• **Throughput**: amount of work that can be done over a period of time
Pipelining doesn’t help latency of single task, it helps throughput of entire workload.

- **Multiple** tasks operating simultaneously using different resources.
- Potential speedup = Number pipe stages.
- Time to “fill” pipeline and time to “drain” it reduces speedup: 2.3X v. 4X in this example.

Diagram:

- **Task Orders**
  - Task A
  - Task B
  - Task C
  - Task D

- **Time**
  - 6 PM
  - 7
  - 8
  - 9

- **Processors**
  - Each task is processed by different resources.

- **Time to **fill** pipeline and time to **drain** it reduces speedup:**
• Suppose new Washer takes 20 minutes, new Stasher takes 20 minutes. How much faster is pipeline?

• Pipeline rate limited by slowest pipeline stage

• Unbalanced lengths of pipe stages reduces speedup
Steps in Executing MIPS

1) **IFtch**: Instruction Fetch, Increment PC

2) **Dcd**: Instruction Decode, Read Registers

3) **Exec**:  
   Mem-ref: Calculate Address  
   Arith-log: Perform Operation

4) **Mem**:  
   Load: Read Data from Memory  
   Store: Write Data to Memory

5) **WB**: Write Data Back to Register
Pipeline Hazard: Matching socks in later load

- A depends on D; **stall** since folder tied up
Administrivia

- HW8 due tomorrow
- Project 2 due next Monday
- Newsgroup problems
- Reminder: Midterm regrades due today
Problems for Pipelining CPUs

• Limits to pipelining: **Hazards** prevent next instruction from executing during its designated clock cycle
  
  - **Structural hazards**: HW cannot support some combination of instructions (single person to fold and put clothes away)
  
  - **Control hazards**: Pipelining of branches causes later instruction fetches to wait for the result of the branch
  
  - **Data hazards**: Instruction depends on result of prior instruction still in the pipeline (missing sock)

• These might result in pipeline stalls or “bubbles” in the pipeline.
Read same memory twice in same clock cycle

Structural Hazard #1: Single Memory (1/2)
Structural Hazard #1: Single Memory (2/2)

• Solution:
  • infeasible and inefficient to create second memory
  • (We’ll learn about this more next week)
  • so simulate this by having two Level 1 Caches (a temporary smaller [of usually most recently used] copy of memory)
  • have both an L1 Instruction Cache and an L1 Data Cache
  • need more complex hardware to control when both caches miss
Can we read and write to registers simultaneously?
Structural Hazard #2: Registers (2/2)

- Two different solutions have been used:
  1) RegFile access is *VERY* fast: takes less than half the time of ALU stage
     - Write to Registers during first half of each clock cycle
     - Read from Registers during second half of each clock cycle
  2) Build RegFile with independent read and write ports

- Result: can perform Read and Write during same clock cycle
1) Thanks to pipelining, I have **reduced the time** it took me to wash my one shirt.

2) Longer pipelines are **always a win** (since less work per stage & a faster clock).

   a) FF
   b) FT
   c) TF
   d) TT
Things to Remember

• Optimal Pipeline
  • Each stage is executing part of an instruction each clock cycle.
  • One instruction finishes during each clock cycle.
  • On average, execute far more quickly.

• What makes this work?
  • Similarities between instructions allow us to use same stages for all instructions (generally).
  • Each stage takes about the same amount of time as all others: little wasted time.
Bonus slides

• These are extra slides that used to be included in lecture notes, but have been moved to this, the “bonus” area to serve as a supplement.

• The slides will appear in the order they would have in the normal presentation.
The Single Cycle Datapath during Jump

- **New PC = { PC[31..28], target address, 00 }**

- Jump = 1
- nPC_sel = ?

- RegDst = x
- RegWr = 0

- J-type

  - op
  - target address
  - jump

- Instruction Fetch Unit
  - Instruction<31:0>

- Instruction Memory
  - Imm16
  - TA26
  - MemtoReg = x

- ALU
  - ALUctr = x
  - ALUSrc = x

- Data Memory
  - MemWr = 0

- Registers
  - 32 32-bit Registers

- Extender
  - ExtOp = x
Instruction Fetch Unit at the End of Jump

• New PC = { PC[31..28], target address, 00 }

How do we modify this to account for jumps?
Instruction Fetch Unit at the End of Jump

**New PC = \{ PC[31..28], target address, 00 \}**

**Query**
- Can Zero still get asserted?
- Does nPC_sel need to be 0?
  - If not, what?