CS 61C:
Great Ideas in Computer Architecture
Building Blocks for Datapaths

Instructor:
Randy H. Katz
http://inst.eecs.Berkeley.edu/~cs61c/fa13

• Parallel Requests
  Assigned to computer
  e.g., Search “Katz”

• Parallel Threads
  Assigned to core
  e.g., Lookup, Ads

• Parallel Instructions
  >1 instruction @ one time
  e.g., 5 pipelined instructions

• Parallel Data
  >1 data item @ one time
  e.g., Add of 4 pairs of words

• Hardware descriptions
  All gates @ one time

• Programming Languages

You are Here!

Software

Harness Parallelism & Achieve High Performance

Hardware

Today

Smart Phone

Warehouse Scale Computer

Logic Gates

Computer

Input/Output

Core

Memory

(Cache)

A0 + B0, A1 + B1, A2 + B2, A3 + B3
Levels of Representation/Interpretation

<table>
<thead>
<tr>
<th>High Level Language Program (e.g., C)</th>
<th>Compiler</th>
</tr>
</thead>
<tbody>
<tr>
<td>Assembly Language Program (e.g., MIPS)</td>
<td>Assembler</td>
</tr>
<tr>
<td>Machine Language Program (MIPS)</td>
<td></td>
</tr>
</tbody>
</table>

```
temp = v[k];
v[k] = v[k+1];
v[k+1] = temp;
```

```
lw $t0, 0($2)
lw $t1, 4($2)
sw $t1, 0($2)
sw $t0, 4($2)
```

High Level Language Program (e.g., C)

Assembly Language Program (e.g., MIPS)

Machine Language Program (MIPS)

Review: The MIPS-lite Subset

- **ADDU and SUBU**
  - addu rd, rs, rt
  - subu rd, rs, rt

- **OR Immediate:**
  - ori rt, rs, imm16

- **LOAD and STORE Word**
  - lw rt, rs, imm16
  - sw rt, rs, imm16

- **BRANCH:**
  - beq rs, rt, imm16

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>26</td>
<td>21</td>
<td>16</td>
<td>11</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>6 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>26</td>
<td>21</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>26</td>
<td>21</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>immediate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>31</td>
<td>26</td>
<td>21</td>
</tr>
<tr>
<td>6 bits</td>
<td>5 bits</td>
<td>5 bits</td>
<td>16 bits</td>
</tr>
</tbody>
</table>
Review: Register Transfer Language (RTL)

- RTL gives the **meaning** of the instructions
  \[
  \{ \text{op, rs, rt, rd, shamt, funct} \} \leftarrow \text{MEM[PC]}
  \]
  \[
  \{ \text{op, rs, rt, Imm16} \} \leftarrow \text{MEM[PC]}
  \]
- All start by fetching the instruction

  **Inst**  **Register Transfers**

  **ADDU**  \( R[rd] \leftarrow R[rs] + R[rt]; \text{PC} \leftarrow \text{PC} + 4 \)

  **SUBU**  \( R[rd] \leftarrow R[rs] - R[rt]; \text{PC} \leftarrow \text{PC} + 4 \)

  **ORI**  \( R[rt] \leftarrow R[rs] | \text{zero_ext(Imm16)}; \text{PC} \leftarrow \text{PC} + 4 \)

  **LOAD**  \( R[rt] \leftarrow \text{MEM}[R[rs] + \text{sign_ext(Imm16)}]; \text{PC} \leftarrow \text{PC} + 4 \)

  **STORE**  \( \text{MEM}[R[rs] + \text{sign_ext(Imm16)}] \leftarrow R[rt]; \text{PC} \leftarrow \text{PC} + 4 \)

  **BEQ**  if ( \( R[rs] == R[rt] \) )

  then \( \text{PC} \leftarrow \text{PC} + 4 + (\text{sign_ext(Imm16)} || 00) \)

  else \( \text{PC} \leftarrow \text{PC} + 4 \)

---

**Agenda**

- MIPS-lite Datapath
- CPU Timing
- MIPS-lite Control
- And, in Conclusion, ...
Agenda

- MIPS-lite Datapath
- CPU Timing
- MIPS-lite Control
- And, in Conclusion, ...

Processor Design Process

- Five 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 1: Requirements of the Instruction Set

- Memory (MEM)
  - Instructions & data (will use one for each: really caches)
- Registers (R: 32 x 32)
  - Read rs
  - Read rt
  - Write rt or rd
- PC
- Extender (sign/zero extend)
- Add/Sub/OR unit for operation on register(s) or extended immediate
- Add 4 (+ maybe extended immediate) to PC
- Compare if registers equal?

Generic Steps of Datapath
Step 2: Components of the Datapath

- Combinational Elements
- State Elements + Clocking Methodology
- Building Blocks

**Adder**

```
A  32'
B  32'
CARRY IN

SUM

CARRY OUT
```

**Multiplexer**

```
SELECT

A  32'
B  32'

Y  32'
```

**ALU**

```
OP

A  32'
B  32'

RESULT
```

ALU Needs for MIPS-lite + Rest of MIPS

- Addition, subtraction, logical OR, ==:
  
  ```
  ADDU R[rd] = R[rs] + R[rt]; ...
  SUBU R[rd] = R[rs] - R[rt]; ...
  ORI R[rt] = R[rs] | zero_ext(Imm16)...
  BEQ if ( R[rs] == R[rt] )...
  ```

- Test to see if output == 0 for any ALU operation gives == test. How?
- P&H also adds AND, Set Less Than (1 if A < B, 0 otherwise)
- ALU from Appendix C, section C.5
Storage Element: Idealized Memory

- Memory (idealized)
  - One input bus: Data In
  - One output bus: Data Out
- Memory word is found by:
  - Address selects the word to put on Data Out
  - Write Enable = 1: address selects the memory word to be written via the Data In bus
- Clock input (CLK)
  - CLK input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block: Address valid ⇒ Data Out valid after “access time”

Storage Element: Register (Building Block)

- Similar to D Flip Flop except
  - N-bit input and output
  - Write Enable input
- Write Enable:
  - Negated (or deasserted) (0): Data Out will not change
  - Asserted (1): Data Out will become Data In on rising edge of clock
Storage Element: Register File

- Register File consists of 32 registers:
  - Two 32-bit output busses: busA and busB
  - One 32-bit input bus: busW
- Register is selected by:
  - RA (number) selects the register to put on busA (data)
  - RB (number) selects the register to put on busB (data)
  - RW (number) selects the register to be written via busW (data) when Write Enable is 1
- Clock input (clk)
  - Clk input is a factor ONLY during write operation
  - During read operation, behaves as a combinational logic block:
    - RA or RB valid ⇒ busA or busB valid after “access time.”

Step 3: Assemble DataPath Meeting Requirements

- Register Transfer Requirements ⇒ Datapath Assembly
- Instruction Fetch
- Read Operands and Execute Operation
- Common RTL operations
  - Fetch the Instruction: mem[PC]
  - Update the program counter:
    - Sequential Code: PC ← PC + 4
    - Branch and Jump: PC ← “something else”
Step 3: Add & Subtract

- \( R[rd] = R[rs] \) op \( R[rt] \) (addu \( rd, rs, rt \))
  
  - \( Ra, Rb, \) and \( Rw \) come from instruction’s \( Rs, Rt, \) and \( Rd \) fields

  \[
  \begin{array}{ccccccc}
  \text{op} & 26 & 21 & 16 & 11 & 6 & 0 \\
  \text{rs} & 5 & 5 & 5 & 5 & 6 & \text{bits} \\
  \text{rt} & 5 & 5 & 5 & 5 & 6 & \text{bits} \\
  \text{rd} & 5 & 5 & 5 & 5 & 6 & \text{bits} \\
  \text{shamt} & \text{bits} \\
  \text{funct} & \text{bits} \\
  \end{array}
  \]

  - \( ALUctr \) and \( RegWr \): control logic after decoding the instruction

- ... Already defined the register file & ALU
Clocking Methodology

- Storage elements clocked by same edge
- "Critical path" (longest path through logic) determines length of clock period
- Have to allow for Clock-to-Q and Setup Times too
- This lecture (and P&H sections) 4.3-4.4 do whole instruction in 1 clock cycle for pedagogic reasons
  - Project 4 will do it in 2 clock cycles via simple pipelining
  - Soon explain pipelining and use 5 clock cycles per instruction

Register-Register Timing: One Complete Cycle

- Clk
- Clk-to-Q
- PC
- Old Value
- New Value
- RegFile
- busA
- busB
- RegWr
- Rd
- Rs
- Rt
- ALUctr
- Setup Time
- Register Write Occurs Here

11/5/13
Administrivia

- HW #6 Due Sunday, 10 November
- Project #4 due Sunday, 17 November
  - Logisim labs posted: #10, #11
  - You can do all of this at home!

Agenda

- MIPS-lite Datapath
- CPU Timing
- MIPS-lite Control
- And, in Conclusion, ...
Register-Register Timing: One Complete Cycle

Logical Operations with Immediate

- \( R[rt] = R[rs] \text{ op } \text{ZeroExt}[\text{imm}16] \)

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{immediate} \\
31 & 26 & 21 & 16 & 15 & 0 \\
\end{array}
\]

But we’re writing to \( Rt \) register??
And immediate ALU input??
Logical Operations with Immediate

- \( R[rt] = R[rs] \text{ op } \text{ZeroExt}[\text{imm16}] \)

\[
\begin{array}{ccccccc}
\text{op} & rs & rt & \text{immediate} \\
\hline
\text{31 bits} & 5 \text{ bits} & 5 \text{ bits} & 16 \text{ bits} & 0 \\
\end{array}
\]

Load Operations

- \( R[rt] = \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] \)

Example: \( \text{lw} \ rt, rs, \text{imm16} \)
Load Operations

- \( R[rt] = \text{Mem}[R[rs] + \text{SignExt}[\text{imm16}]] \)

Example: \( lw \ rt, rs, \text{imm16} \)

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{immediate} \\
6 \text{ bits} & 5 \text{ bits} & 5 \text{ bits} & 16 \text{ bits} \\
\end{array}
\]

### RTL: The Add Instruction

\( \text{add rd, rs, rt} \)

- \( \text{MEM[PC]} \) Fetch the instruction from memory
- \( R[rd] = R[rs] + R[rt] \) The actual operation
- \( PC = PC + 4 \) Calculate the next instruction’s address
Instruction Fetch Unit at Beginning of Add

- Fetch the instruction from Instruction memory:
  Instruction = MEM[PC]
  — same for all instructions

Single Cycle Datapath during Add

R[rd] = R[rs] + R[rt]
Instruction Fetch Unit at End of Add

- PC = PC + 4
  - Same for all instructions except: Branch and Jump

Single Cycle Datapath during OR Immediate

- R[rt] = R[rs] OR ZeroExt[Imm16]
Single Cycle Datapath during **OR** Immediate

- \( R_{rt} = R_{rs} \) OR ZeroExt[Imm16]

Single Cycle Datapath during **Load**

- \( R_{rt} = \text{Data Memory} \{ R_{rs} + \text{SignExt}[\text{imm16}] \} \)
Single Cycle Datapath during Load

- \( R[rt] = \text{Data Memory} \{R[rs] + \text{SignExt}[\text{imm16}]\} \)

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{immediate} \\
31 & 26 & 21 & 16 & 0
\end{array}
\]

Single Cycle Datapath during Store

- Data Memory \( \{R[rs] + \text{SignExt}[\text{imm16}]\} = R[rt] \)

\[
\begin{array}{cccccc}
\text{op} & \text{rs} & \text{rt} & \text{immediate} \\
31 & 26 & 21 & 16 & 0
\end{array}
\]
**Single Cycle Datapath during Store**

- Data Memory \( [R_{rs} + \text{SignExt}(\text{imm16})] = R_{rt} \)

**Single Cycle Datapath during Branch**

- if \((R_{rs} - R_{rt} == 0)\) then \(\text{Zero} = 1\); else \(\text{Zero} = 0\)
Single Cycle Datapath during Branch

- if \( R[rs] - R[rt] == 0 \) then \( \text{Zero} = 1 \); else \( \text{Zero} = 0 \)

Instruction Fetch Unit at the End of Branch

- if \( \text{Zero} == 1 \) then \( \text{PC} = \text{PC} + 4 + \text{SignExt}[\text{imm16}] \times 4 \); else \( \text{PC} = \text{PC} + 4 \)

- What is encoding of \( \text{nPC}_\text{sel} \)?
  - Direct MUX select?
  - Branch inst. / not branch

- Let’s pick 2nd option

Q: What logic gate?
Summary: Datapath’s Control Signals

- **ExtOp:** “zero”, “sign”
- **ALUsrc:** 0 ⇒ regB; 1 ⇒ immed
- **ALUctr:** “ADD”, “SUB”, “OR”
- **MemWr:** 1 ⇒ write memory
- **MemtoReg:** 0 ⇒ ALU; 1 ⇒ Mem
- **RegDst:** 0 ⇒ “rt”; 1 ⇒ “rd”
- **RegWr:** 1 ⇒ write register

Given Datapath: RTL → Control
Summary of the Control Signals (1/2)

### Register Transfer

- **add**
  - $R[rd] ← R[rs] + R[rt]$; PC $←$ PC + 4
  - ALUsrc=RegB, ALUctr="ADD", RegDst=rd, RegWr, nPC_sel="+4"

- **sub**
  - $R[rd] ← R[rs] - R[rt]$; PC $←$ PC + 4
  - ALUsrc=RegB, ALUctr="SUB", RegDst=rd, RegWr, nPC_sel="+4"

- **ori**
  - $R[rt] ← R[rs] + \text{zero\_ext}(\text{Imm16})$; PC $←$ PC + 4
  - ALUsrc=Imm, Exttop="Z", ALUctr="OR", RegDst=rt, RegWr, nPC_sel="+4"

- **lw**
  - $R[rt] ← \text{MEM}[ R[rs] + \text{sign\_ext}(\text{Imm16})]$; PC $←$ PC + 4
  - ALUsrc=Imm, Exttop="sn", ALUctr="ADD", MemtoReg, RegDst=rt, RegWr, nPC_sel="+4"

- **sw**
  - $\text{MEM}[ R[rs] + \text{sign\_ext}(\text{Imm16})] ← R[rs]$; PC $←$ PC + 4
  - ALUsrc=Imm, Exttop="sn", ALUctr="ADD", MemWrite, nPC_sel="+4"

- **beq**
  - if $(R[rs] == R[rt])$ then PC $←$ PC + $\text{sign\_ext}(\text{Imm16}) || 00$
  - else PC $←$ PC + 4
  - nPC_sel = "br", ALUctr = "SUB"

### Summary of the Control Signals (2/2)

<table>
<thead>
<tr>
<th>See Appendix A</th>
<th>func</th>
<th>op</th>
<th>000000</th>
<th>000010</th>
<th>001100</th>
<th>001110</th>
<th>100010</th>
<th>100000</th>
<th>001000</th>
<th>101000</th>
<th>101010</th>
<th>000100</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegDst</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>x</td>
<td>x</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>RegWrite</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>nPCsel</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></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>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></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></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### op

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>add, sub</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### I-type

<table>
<thead>
<tr>
<th>31</th>
<th>26</th>
<th>21</th>
<th>16</th>
<th>11</th>
<th>6</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ori, lw, sw, beq</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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:

\[
\begin{align*}
\text{rtype} &= \neg \text{op}_5 \cdot \neg \text{op}_4 \cdot \neg \text{op}_3 \cdot \neg \text{op}_2 \cdot \neg \text{op}_1 \cdot \text{op}_0 \\
\text{ori} &= \neg \text{op}_5 \cdot \neg \text{op}_4 \cdot \text{op}_3 \cdot \text{op}_2 \cdot \neg \text{op}_1 \cdot \text{op}_0 \\
\text{lw} &= \text{op}_5 \cdot \neg \text{op}_4 \cdot \neg \text{op}_3 \cdot \neg \text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\
\text{sw} &= \text{op}_5 \cdot \neg \text{op}_4 \cdot \text{op}_3 \cdot \neg \text{op}_2 \cdot \text{op}_1 \cdot \text{op}_0 \\
\text{beq} &= \neg \text{op}_5 \cdot \neg \text{op}_4 \cdot \text{op}_3 \cdot \text{op}_2 \cdot \neg \text{op}_1 \cdot \neg \text{op}_0 \\
\text{jump} &= \neg \text{op}_5 \cdot \neg \text{op}_4 \cdot \text{op}_3 \cdot \text{op}_2 \cdot \text{op}_1 \cdot \neg \text{op}_0 \\
\text{add} &= \text{rtype} \cdot \text{func}_1 \cdot \neg \text{func}_2 \cdot \neg \text{func}_3 \cdot \neg \text{func}_4 \cdot \neg \text{func}_5 \cdot \text{func}_6 \\
\text{sub} &= \text{rtype} \cdot \text{func}_1 \cdot \neg \text{func}_2 \cdot \neg \text{func}_3 \cdot \neg \text{func}_4 \cdot \text{func}_5 \cdot \neg \text{func}_6
\end{align*}
\]

How do we implement this in gates?
AND Control in Logisim

OR Control Logic in Logisim
Single Cycle Performance

- Assume time for actions are
  - 100ps for register read or write; 200ps for other events
- Clock rate is?

<table>
<thead>
<tr>
<th>Instr</th>
<th>Instr fetch</th>
<th>Register read</th>
<th>ALU op</th>
<th>Memory access</th>
<th>Register write</th>
<th>Total time</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td>100 ps</td>
<td>800ps</td>
</tr>
<tr>
<td>sw</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td>200ps</td>
<td></td>
<td>700ps</td>
</tr>
<tr>
<td>R-format</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td>100 ps</td>
<td>600ps</td>
</tr>
<tr>
<td>beq</td>
<td>200ps</td>
<td>100 ps</td>
<td>200ps</td>
<td></td>
<td></td>
<td>500ps</td>
</tr>
</tbody>
</table>

- What can we do to improve clock rate?
- Will this improve performance as well?
  - Want increased clock rate to mean faster programs

And in Conclusion, ...

**Single-Cycle Processor**

- Five 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 affects the register transfer.
  5. Assemble the control logic
    - Formulate Logic Equations
    - Design Circuits