Review

• CPU design involves Datapath, Control
  – 5 Stages for MIPS Instructions
    1. Instruction Fetch
    2. Instruction Decode & Register Read
    3. ALU (Execute)
    4. Memory
    5. Register Write

• Datapath timing: single long clock cycle or one short clock cycle per stage
Datapath and Control

- Datapath based on data transfers required to perform instructions
- Controller causes the right transfers to happen
Review CPU Clocking (1/2)

• For each instruction, how do we control the flow of information through the datapath?

• Single Cycle CPU: All stages of an instruction completed within one long clock cycle
  – Clock cycle sufficiently long to allow each instruction to complete all stages without interruption within one cycle
Review CPU Clocking (2/2)

• Alternative multiple-cycle CPU: only one stage of instruction per clock cycle
  – Clock is made as long as the slowest stage

  1. Instruction Fetch
  2. Decode/Register Read
  3. Execute
  4. Memory
  5. Register Write

  – Several significant advantages over single cycle execution:
    Unused stages in a particular instruction can be skipped
    OR instructions can be pipelined (overlapped)
Agenda

• Stages of the Datapath
• Datapath Instruction Walkthroughs
• Datapath Design
Five Components of a Computer

- Processor
- Control
- Datapath
- Memory (passive) (where programs, data live when running)
- Devices
  - Input
  - Output
- Keyboard, Mouse
Disk (where programs, data live when not running)
- Display, Printer

Dr Dan Garcia
Processor Design: 5 steps

Step 1: Analyze instruction set to determine datapath requirements
- Meaning of each instruction is given by register transfers
- Datapath must include storage element for ISA registers
- Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
Processor Design: Step 1

Step 1: Analyze instruction set to determine datapath requirements
- Meaning of each instruction is given by register transfers
- Datapath must include storage element for ISA registers
- Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
The MIPS Instruction Formats

- All MIPS instructions are 32 bits long. 3 formats:
  - **R-type**
    
    | 31 | 26 | 21 | 16 | 11 | 6 | 0 |
    |----|----|----|----|----|----|----|
    | op | rs | rt | rd | shamt | funct |
  
    - 6 bits  
    - 5 bits  
    - 5 bits  
    - 5 bits  
    - 5 bits  
    - 6 bits  

  - **I-type**
    
    | 31 | 26 | 21 | 16 | 0 |
    |----|----|----|----|----|
    | op | rs | rt | address/immediate |
  
    - 6 bits  
    - 5 bits  
    - 5 bits  
    - 16 bits  

  - **J-type**
    
    | 31 | 26 |
    |----|----|
    | op | target address |
  
    - 6 bits  
    - 26 bits  

- The different fields are:
  - **op**: operation (“opcode”) of the instruction
  - **rs, rt, rd**: the source and destination register specifiers
  - **shamt**: shift amount
  - **funct**: selects the variant of the operation in the “op” field
  - **address / immediate**: address offset or immediate value
  - **target address**: target address of jump instruction
## Control signals derived from instruction

### R-type

<table>
<thead>
<tr>
<th></th>
<th>0</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shamt</th>
<th>funct</th>
</tr>
</thead>
</table>

### Load/Store

<table>
<thead>
<tr>
<th></th>
<th>35 or 43</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td>15:0</td>
<td></td>
</tr>
</tbody>
</table>

### Branch

<table>
<thead>
<tr>
<th></th>
<th>4</th>
<th>rs</th>
<th>rt</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>31:26</td>
<td>25:21</td>
<td>20:16</td>
<td>15:0</td>
<td></td>
</tr>
</tbody>
</table>

- **opcode**: always read
- **read, except for load**: write for R-type and load
- **sign-extend and add**:
Datapath With Control
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`
Register Transfer Language (RTL)

RTL gives the meaning of the instructions

All start by fetching the instruction

\{ op , rs , rt , rd , shamt , funct \} \leftarrow \text{MEM}[ \text{PC} ]

\{ op , rs , rt , \text{Imm16} \} \leftarrow \text{MEM}[ \text{PC} ]

<table>
<thead>
<tr>
<th>Inst</th>
<th>Register Transfers</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDU</td>
<td>\text{R[rd]} \leftarrow \text{R[rs]} + \text{R[rt]}; \text{PC} \leftarrow \text{PC} + 4</td>
</tr>
<tr>
<td>SUBU</td>
<td>\text{R[rd]} \leftarrow \text{R[rs]} - \text{R[rt]}; \text{PC} \leftarrow \text{PC} + 4</td>
</tr>
<tr>
<td>ORI</td>
<td>\text{R[rt]} \leftarrow \text{R[rs]} \mid \text{zero_ext(Imm16)}; \text{PC} \leftarrow \text{PC} + 4</td>
</tr>
<tr>
<td>LOAD</td>
<td>\text{R[rt]} \leftarrow \text{MEM[ R[rs] + sign_ext(Imm16) ]}; \text{PC} \leftarrow \text{PC} + 4</td>
</tr>
<tr>
<td>STORE</td>
<td>\text{MEM[ R[rs] + sign_ext(Imm16) ]} \leftarrow \text{R[rt]}; \text{PC} \leftarrow \text{PC} + 4</td>
</tr>
<tr>
<td>BEQ</td>
<td>\text{if ( R[rs] == R[rt] )}</td>
</tr>
<tr>
<td></td>
<td>\text{then PC} \leftarrow \text{PC} + 4 + (\text{sign_ext(Imm16)} \mid\mid 00)</td>
</tr>
<tr>
<td></td>
<td>\text{else PC} \leftarrow \text{PC} + 4</td>
</tr>
</tbody>
</table>
Step 1: Requirements of the Instruction Set

- Memory (MEM)
  - Instructions & data (will use one for each)
- 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 registers?
Processor Design: Step 2

Step 1: Analyze instruction set to determine datapath requirements
- Meaning of each instruction is given by register transfers
- Datapath must include storage element for ISA registers
- Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
Step 2: Components of the Datapath

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

Adder

Multiplexer

ALU

Dr Dan Garcia
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 \( R[rt] = R[rs] \mid \text{zero_ext}(\text{Imm16}); \) ...
  
  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 follows Chapter 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 $\Rightarrow$ 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 positive 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 $\Rightarrow$ busA or busB valid after “access time.”
Clocking Methodology

- Storage elements clocked by same edge
- Flip-flops (FFs) and combinational logic have some delays
  - Gates: delay from input change to output change
  - Signals at FF D input must be stable before active clock edge to allow signal to travel within the FF (set-up time), and we have the usual clock-to-Q delay
- “Critical path” (longest path through logic) determines length of clock period

Dr Dan Garcia
Register-Register Timing:
One Complete Cycle (Add/Sub)

- Clk
- PC
- Rs, Rt, Rd,
- Op, Func
- ALUctr
- RegWr
- busA, B
- busW
- RegFile
- ALU

Register Write Occurs Here

Dr Dan Garcia
## Register-Register Timing: One Complete Cycle

### Clk
- **Old Value**
- **New Value**

### PC
- **Old Value**
- **New Value**

### Rs, Rt, Rd, Op, Func
- **Old Value**
- **New Value**

### ALUctr
- **Old Value**
- **New Value**

### RegWr
- **Old Value**
- **New Value**

### busA, B
- **Old Value**
- **New Value**

### busW
- **Old Value**
- **New Value**

**Instruction Memory Access Time**
- Old Value
- New Value

**Delay through Control Logic**
- Old Value
- New Value

**Register File Access Time**
- Old Value
- New Value

**ALU Delay**
- Old Value
- New Value

**Register Write Occurs Here**

---

Dr Dan Garcia
Register-Register Timing: One Complete Cycle

Clk +---------------------------------------------------+
PC | Old Value | New Value |
Rs, Rt, Rd, Op, Func | Old Value | New Value |
ALUctr | Old Value | New Value |
RegWr | Old Value | New Value |
busA, B | Old Value | New Value |
busW | Old Value | New Value |

Instruction Memory Access Time
Delay through Control Logic
Register File Access Time
ALU Delay

RegWr: Rd, Rs, Rt
ALUctr: busA (32), busB (32)
RegFile: Rw, Ra, Rb

Register Write Occurs Here

Dr Dan Garcia
Processor Design: Step 3

Step 1: Analyze instruction set to determine datapath requirements
- Meaning of each instruction is given by register transfers
- Datapath must include storage element for ISA registers
- Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
Putting it All Together: A Single Cycle Datapath

Diagram of a single cycle datapath with various components and signals:
- Instruction Memory (Inst Memory)
- RegFile
- ALU
- Data Memory

Signals and components:
- Instruction<31:0>
- Rs, Rt, Rd, Imm16
- RegDst, RegWr, MemtoReg, MemWr
- ALUctrl
- Equal
- ALUSrc, ExtOp
- WrEn, Addr
- Data In
- clk
- nPC_sel, PC
- Adder, Mux
- busA, busB
- 32-bit signals and values
Peer Instruction

• Our ALU is synchronous device?
• We should use the main ALU to calculate \( PC = PC + 4 \) ?
• The ALU is inactive for memory reads or writes ?
Processor Design: Step 4

Step 1: Analyze instruction set to determine datapath requirements
- Meaning of each instruction is given by register transfers
- Datapath must include storage element for ISA registers
- Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
Step 3a: Instruction Fetch Unit

- 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 3b: Add & Subtract

- \( R[rd] = R[rs] \text{ op } R[rt] \) (addu rd, rs, rt)
  - Ra, Rb, and Rw come from instruction’s Rs, Rt, and Rd fields
  - \( 32 \times 32 \)-bit Registers

- ALUctr and RegWr: control logic after decoding the instruction

- ... Already defined the register file & ALU
R-Type Instruction
3c: Logical Op (or) with Immediate

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

**Writing to Rt register (not Rd)!!**

**What about Rt Read?**
3d: Load Operations

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

Example: \( \text{lw} \ rt, rs, \text{imm16} \)
3d: Load Operations

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

Example: \( lw \ rt, rs, imm16 \)
3e: Store Operations


Ex.: \texttt{sw rt, rs, imm16}

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

RegDst \rightarrow Rd, Rt

RegWr \rightarrow Rs, Rt

MemtoReg

ALUctr

MemWr

RegFile

ExtOp

ALUSrc

Extender

Data Memory

Dr Dan Garcia
3e: Store Operations


Example: `sw rt, rs, imm16`

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

Dr Dan Garcia
Load Instruction
3f: The Branch Instruction

beq rs, rt, imm16

- mem[PC] Fetch the instruction from memory
- Equal = R[rs] == R[rt] Calculate branch condition
- if (Equal) Calculate the next instruction’s address
  - PC = PC + 4 + ( SignExt(imm16) x 4 )
else
  - PC = PC + 4
Datapath for Branch Operations

beq rs, rt, imm16

Datapath generates condition (Equal)

Inst Address

Already have mux, adder, need special sign extender for PC, need equal compare (sub?)
Instruction Fetch Unit including Branch

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

- How to encode nPC_sel?
- Direct MUX select?
- Branch inst. / not branch inst.
- Let’s pick 2nd option

Q: What logic gate?

<table>
<thead>
<tr>
<th>nPC_sel</th>
<th>zero?</th>
<th>MUX</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>x</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>
Putting it All Together (without jump) : A Single Cycle Datapath

- **Inst Memory**
  - Instruction<31:0>
  - Rs, Rt, Rd, Imm16

- **RegFile**
  - Rw, Ra, Rb
  - busA, busB

- **ALU**
  - ALUctr
  - Equal
  - WrEn, Adr
  - Data Memory

- **Extender**
  - ExtOp
  - ALUSrc

- **PC Ext**
  - 0

- **Adder**
  - clk

- **Mux**
  - 00, 1

- **RegDst**
  - Rd, Rt

- **RegWr**
  - Rs, Rt

- **Data In**
  - 00

- **MemtoReg**
  - MemWr

- **nPC_sel**
  - 4

- **imm16**

---

Dr Dan Garcia
Implementing Jumps

- Jump uses word address
- Update PC with concatenation of
  - Top 4 bits of old PC
  - 26-bit jump address
  - 00
- Need an extra control signal decoded from opcode
Single Cycle Datapath during Jump

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

Dr Dan Garcia
Single Cycle Datapath during Jump

- New PC = \{ PC[31..28], target address, 00 \}
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?
Datapath With Jumps Added
Datapath 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 $\rightarrow$ Control

Instruction$^{31:0}$

<table>
<thead>
<tr>
<th>Inst Memory</th>
<th>Addr</th>
</tr>
</thead>
</table>

| $^{26:31}$ | $^{21:25}$ | $^{16:20}$ | $^{11:15}$ | $^{0:15}$ |
| Op | Fun | Rt | Rs | Rd | Imm16 |

Control

nPC_sel RegWr RegDst ExtOp ALUSrc ALUctr MemWr MemtoReg

DATA PATH
RTL: The Add Instruction

- **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

The Add Instruction

<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>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>
Instruction Fetch Unit at the Beginning of Add

- Fetch the instruction from Instruction memory: Instruction = MEM[PC]
  - same for all instructions
Single Cycle Datapath during Add

$R[rd] = R[rs] + R[rt]$
• $PC = PC + 4$
  
  – Same for all instructions except: Branch and Jump

Dr Dan Garcia
Processor Design: Step 5

Step 1: Analyze instruction set to determine datapath requirements
– Meaning of each instruction is given by register transfers
– Datapath must include storage element for ISA registers
– Datapath must support each register transfer

Step 2: Select set of datapath components & establish clock methodology

Step 3: Assemble datapath components that meet the requirements

Step 4: Analyze implementation of each instruction to determine setting of control points that realizes the register transfer

Step 5: Assemble the control logic
## 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>$R[rd] \leftarrow R[rs] + R[rt]; \ PC \leftarrow PC + 4$</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>$R[rd] \leftarrow R[rs] - R[rt]; \ PC \leftarrow PC + 4$</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>$R[rt] \leftarrow R[rs] + \text{zero}_\text{ext}(\text{Imm16}); \ PC \leftarrow PC + 4$</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>$R[rt] \leftarrow \text{MEM}[ R[rs] + \text{sign}_\text{ext}(\text{Imm16})]; \ PC \leftarrow PC + 4$</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>$\text{MEM}[ R[rs] + \text{sign}_\text{ext}(\text{Imm16})] \leftarrow R[rs]; \ PC \leftarrow PC + 4$</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 $PC \leftarrow PC + \text{sign}_\text{ext}($Imm16$) \land 00$</td>
</tr>
<tr>
<td></td>
<td>else $PC \leftarrow PC + 4$</td>
</tr>
<tr>
<td></td>
<td>nPC_sel = “br”, ALUctr = “SUB”</td>
</tr>
</tbody>
</table>
### Summary of the Control Signals (2/2)

See Appendix A

<table>
<thead>
<tr>
<th></th>
<th>func op 10 0000</th>
<th>func op 10 0010</th>
<th>(xx...x) [We Don't Care]</th>
</tr>
</thead>
<tbody>
<tr>
<td>RegDst</td>
<td>add 1</td>
<td>sub 1</td>
<td>x</td>
</tr>
<tr>
<td>ALUSrc</td>
<td>sub 0</td>
<td>ori 0</td>
<td>sw 1</td>
</tr>
<tr>
<td>MemtoReg</td>
<td>0 1</td>
<td>lw 0</td>
<td>beq 1</td>
</tr>
<tr>
<td>RegWrite</td>
<td>1 1</td>
<td>sw 0</td>
<td>jump x</td>
</tr>
<tr>
<td>MemWrite</td>
<td>0 0</td>
<td>lw 0</td>
<td>jump x</td>
</tr>
<tr>
<td>nPCsel</td>
<td>0 0</td>
<td>lw 0</td>
<td>jump x</td>
</tr>
<tr>
<td>Jump</td>
<td>0 1</td>
<td>lw 0</td>
<td>jump x</td>
</tr>
<tr>
<td>ExtOp</td>
<td>x x</td>
<td>lw 0</td>
<td>jump x</td>
</tr>
<tr>
<td>ALUctr&lt;2:0&gt;</td>
<td>0 0</td>
<td>lw 0</td>
<td>jump x</td>
</tr>
</tbody>
</table>

### Operands

- **R-type**
  - `op`: add, sub
  - `rs`, `rt`, `rd`: ori, lw, sw, beq
  - `shamt`, `funct`: target address

- **I-type**
  - `op`: ori, lw, sw, beq
  - `rs`, `rt`: target address

- **J-type**
  - `op`: jump
## ALU Control

- Assume 2-bit ALUOp derived from opcode
- Combinational logic derives ALU control

<table>
<thead>
<tr>
<th>opcode</th>
<th>ALUOp</th>
<th>Operation</th>
<th>funct</th>
<th>ALU function</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw</td>
<td>00</td>
<td>load word</td>
<td>XXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>sw</td>
<td>00</td>
<td>store word</td>
<td>XXXXXX</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>branch equal</td>
<td>XXXXXX</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td>R-type</td>
<td>10</td>
<td>add</td>
<td>100000</td>
<td>add</td>
<td>0010</td>
</tr>
<tr>
<td></td>
<td></td>
<td>subtract</td>
<td>100010</td>
<td>subtract</td>
<td>0110</td>
</tr>
<tr>
<td></td>
<td></td>
<td>AND</td>
<td>100100</td>
<td>AND</td>
<td>0000</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OR</td>
<td>100101</td>
<td>OR</td>
<td>0001</td>
</tr>
<tr>
<td></td>
<td></td>
<td>set-on-less-than</td>
<td>101010</td>
<td>set-on-less-than</td>
<td>0111</td>
</tr>
</tbody>
</table>
Boolean Expressions for Controller

RegDst  = add + sub
ALUSrc  = ori + lw + sw
MemtoReg = lw
RegWrite = add + sub + ori + lw
MemWrite = sw
nP Cs sel = beq
Jump = jump
ExtOp = lw + sw
ALUctr[0] = sub + beq  (assume ALUctr is 00 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

How do we implement this in gates?

Dr Dan Garcia
Controller Implementation

“AND” logic

opcode

func

add

sub

ori

lw

sw

beq

jump

“OR” logic

RegDst

ALUSrc

MemtoReg

RegWrite

MemWrite

nPcSel

Jump

ExtOp

ALUctr[0]

ALUctr[1]
Summary: 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 effect the register transfer.
5. Assemble the control logic
   • Formulate Logic Equations
   • Design Circuits