Different Implementations

- **Multi-cycle** MIPS, multiple CPI:
  http://aggregate.org/EE380/multiv.html

- **Single-cycle**, 1 CPI, but slow clock:
  http://aggregate.org/EE380/onebeq.html

- **Pipelined**, multiple CPI, but fast clock and *throughput* up to 1 instruction/cycle
Single-Cycle MIPS Design

- Process one instruction at a time...
- Here’s a **multi-cycle** MIPS:
  
  http://aggregate.org/EE380/multiv.html
  
  - One instruction, **multiple clock cycles**
  - **Minimal HW**, state machine control

- Our **single-cycle** MIPS design:
  - Each instruction takes **one clock cycle**
  - Lots of hardware, and **not very fast**...
  - **No state machine in control logic**
Single-Cycle MIPS Design

- This is **not** a design you’d really build…
  it’s a step toward a **pipelined design**

- We’re not going to design it all at once
  - Can **incrementally design & test**
  - Start by implementing one instruction
  - Handle an instruction sequence
  - Keep adding instructions…

- **Learn the process**, not the design
Types Of Things In MIPS

- Want consistent attributes across all parts of the design, e.g., word size
- Easier to debug and maintain abstracted; e.g., `reg [31:0] t;` holds a word or address?

```plaintext
// Types
`define WORD    [31:0] // size of a data word
`define ADDR    [31:0] // size of a memory address
`define INST    [31:0] // size of an instruction
`define REG     [4:0]  // size of a register number
`define REGCNT  [31:0] // register count
`define MEMCNT  [511:0] // memory count implemented
`define OPCODE  [5:0]  // 6-bit opcodes
```
MIPS Instruction Fields

// Fields
`define OP [31:26] // opcode field
`define RS [25:21] // rs field
`define RT [20:16] // rt field
`define RD [15:11] // rd field
`define IMM [15:0] // immediate/offset field
`define SHAMT [10:6] // shift amount
`define FUNCT [5:0] // function code (opcode extension)
`define JADDR [25:0] // jump address field
Let’s start with addu

```
addu $rd,$rs,$rt
```
Now to addu

- The addu $rd,$rs,$rt instruction:
  - Fetch the instruction
  - Read values from registers $rs and $rt
  - Add them
  - Write result into $rd

assign ir = m[pc];

always @(posedge clk) begin
  r[ir `RD] <= r[ir `RS] + r[ir `RT];
  halt <= 1;
end
The Processor Testbench

// Testbench options
`define RUNTIME 100       // How long simulator can run
`define CLKDEL 1         // Clock transition delay

// Testbench
module bench;
reg reset = 1; reg clk = 0; wire halt;
processor PE(halt, reset, clk);
initial begin
    #`CLKDEL clk = 1;
    #`CLKDEL clk = 0;
    reset = 0;
    while (($time < `RUNTIME) && !halt) begin
        #`CLKDEL clk = 1;
        #`CLKDEL clk = 0;
    end
end
endmodule
How do we know it works?

• Need to put an instruction in memory

```
`define JPACK(R,O,J) begin R`OP=0; R`JADDR=J; end
`define IPACK(R,O,S,T,I) begin R`OP=0; R`RS=S; R`RT=T; \n  R`IMM=I; end
`define RPACK(R,S,T,D,SH,FU) begin R`OP=`RTYPE; R`RS=S; \n  R`RT=T; R`RD=D; R`SHAMT=SH; R`FUNCT=FU; end
initial `RPACK(m[0], 2, 3, 1, 0, `ADDU);
```

• Need some way to see what happens

```
`define TRACE 1 // enable simulation trace

`ifdef TRACE
  $display(...);
`endif
```
Complete MIPS Processor*

*that only understands one addu

• Can run it here:
  http://aggregate.org/EE380/oneaddu.html

• Impressed?
Complete MIPS Processor*

*that only understands one addu

• Can run it here:
  http://aggregate.org/EE380/oneaddu.html

• Impressed?

• OK, how about we make it understand how to execute a sequence of addu?
addu $rd,$rs,$rt ...
To execute a sequence of addu

- We need to build the PC incrementer:
  ```
  assign PCAdd = (pc + 4);
  ```

- We also need to check for a legal instruction:
  ```
  if ((ir `OP != `RTYPE) || (ir `FUNCT != `ADDU)) ...
  ```

- Can run it here:
  [http://aggregate.org/EE380/oneaddus.html](http://aggregate.org/EE380/oneaddus.html)
Other RTYPE Instructions?

- There are lots of instructions like `addu`:
  
<table>
<thead>
<tr>
<th>Instruction</th>
<th>Syntax</th>
<th>Equation</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>addu</code></td>
<td><code>$rd, $rs, $rt</code></td>
<td><code>$rd = $rs + $rt</code></td>
</tr>
<tr>
<td><code>sltu</code></td>
<td><code>$rd, $rs, $rt</code></td>
<td><code>$rd = $rs &lt; $rt</code></td>
</tr>
<tr>
<td><code>and</code></td>
<td><code>$rd, $rs, $rt</code></td>
<td><code>$rd = $rs &amp; $rt</code></td>
</tr>
<tr>
<td><code>or</code></td>
<td><code>$rd, $rs, $rt</code></td>
<td>`$rd = $rs</td>
</tr>
<tr>
<td><code>xor</code></td>
<td><code>$rd, $rs, $rt</code></td>
<td><code>$rd = $rs ^ $rt</code></td>
</tr>
<tr>
<td><code>subu</code></td>
<td><code>$rd, $rs, $rt</code></td>
<td><code>$rd = $rs - $rt</code></td>
</tr>
</tbody>
</table>

- Handle them all the same, except in ALU
addu $rd,$rs,$rt  
subu $rd,$rs,$rt  
slt $rd,$rs,$rt  
and $rd,$rs,$rt  
or $rd,$rs,$rt  
xor $rd,$rs,$rt
Decoding RTYPE Inst.

// Decode OP, FUNCT into one 7-bit EXTOP
module decode(xop, ir);
output reg `EXTOP xop; // decoded 7-bit op
input `INST ir; // instruction

always @(ir) begin
  case (ir `OP)
    `RTYPE: case (ir `FUNCT)
      `ADDU, `SUBU,
      `AND, `OR, `XOR,
      `SLTU: xop = `F(ir `FUNCT);
      default: xop = `TRAP; // trap illegal instruction
    endcase
    default: xop = `TRAP; // trap illegal instruction
  endcase
end
endmodule
module alu(res, xop, top, bot);
output reg `WORD res; // combinatorial result
input `EXTOP xop; // extended operation
input `WORD top, bot; // top & bottom inputs

always @(xop or top or bot) begin
  case (xop)
    `F(`ADDU): res = (top + bot);
    `F(`SLTU): res = (top < bot);
    `F(`AND): res = (top & bot);
    `F(`OR): res = (top | bot);
    `F(`XOR): res = (top ^ bot);
    `F(`SUBU): res = (top - bot);
  default: res = top;
  endcase
end
endmodule
Handle All RTYPE Inst.

• There’s a bit of wiring to tie stuff together...

// Function unit wiring
wire `ADDR PCAdd;
wire `EXTOP ALUcontrol;
wire `WORD ALUresult;

// Control logic
assign ALUOp = (ir `OP);

// Function units
decode DECODE(ALUcontrol, ir);
alu    ALU(ALUresult, ALUcontrol, r[ir `RS], r[ir `RT]);

• Can run it here:
  http://aggregate.org/EE380/onertypes.html
How About Immediates?

• Immediates use the ALU similarly...

```
addiu  $rt,$rs,imm  $rt = $rs + imm
sltiu $rt,$rs,imm  $rt = $rs < imm
andi   $rt,$rs,imm  $rt = $rs & imm
ori    $rt,$rs,imm  $rt = $rs | imm
xori   $rt,$rs,imm  $rt = $rs ^ imm
lui    $rt,imm      $rt = imm << 16
```

• Lui is odd, but is encoded as $0 for $rs
Immediate Data Paths

• Need immediate sign extender for 16 to 32 bits:

wire `WORD Signextend;
assign Signextend = {{16{ir[15]}}, ir `IMM};

• Need muxes to select ALU inputs, reg to write:

wire ALUSrc;
wire `REG RegDstMux;
wire `WORD ALUSrcMux;
assign RegDst = (ALUOp == `RTYPE);
assign RegDstMux = (RegDst ? ir `RD : ir `RT);
assign ALUSrcMux = (ALUSrc ? Signextend : r[ir `RT]);
Decoding Imm Inst.

```verilog
// Decode OP, FUNCT into one 7-bit EXTOP
module decode(xop, ir);
output reg `EXTOP xop; // decoded 7-bit op
input `INST ir;      // instruction

always @(ir) begin
    case (ir `OP)
        `RTYPE: case (ir `FUNCT)
            `ADDU, `SUBU,
            `AND, `OR, `XOR,
            `SLTU:    xop = `F(ir `FUNCT);
            default: xop = `TRAP; // trap illegal instruction
        endcase
        `ADDIU, `SLTIU,
        `ANDI, `ORI, `XORI,
        `LUI:     xop = ir `OP;
        default:  xop = `TRAP; // trap illegal instruction
    endcase end
endmodule
```
ALU for Imm Instructions

// General-purpose ALU
module alu(res, xop, top, bot);
output reg `WORD res;    // combinatorial result
input `EXTOP xop;       // extended operation
input `WORD top, bot;   // top & bottom inputs

// combinatorial always using sensitivity list
// output declared as reg, but never use <=
always @(xop or top or bot) begin
    case (xop)
        `ADDIU, `F(`ADDU): res = (top + bot);
        `SLTIU, `F(`SLTU): res = (top < bot);
        `ANDI, `F(`AND): res = (top & bot);
        `ORI, `F(`OR): res = (top | bot);
        `XORI, `F(`XOR): res = (top ^ bot);
        `LUI:                  res = (bot <<< 16);
        `F(`SUBU): res = (top - bot);
        // should always cover all possible values
        default: res = top;
    endcase
endendmodule
RTYPE & Immediate...

• Of course, we add new test cases:
  `IPACK(m[6], `ADDIU, 3, 1, -1);
  `IPACK(m[7], `SLTIU, 5, 1, 12345);
  `IPACK(m[8], `ANDI, 3, 1, 3);
  `IPACK(m[9], `ORI, 3, 1, 3);
  `IPACK(m[10], `XORI, 3, 1, 3);
  `IPACK(m[11], `LUI, 0, 1, 1);

• The TRACE output had to be upgraded:

```plaintext
if (ir `OP) $display("%d: OP=%x RS=%d RT=%d IMM=%x", pc, ir `OP, ir `RS, ir `RT, ir `IMM); else
  $display("%d: OP=%x RS=%d RT=%d RD=%d SHAMT=%d FUNCT=%x", pc, ir `OP, ir `RS, ir `RT, ir `RD, ir `SHAMT, ir `FUNCT);
```

• Can run it here:
  http://aggregate.org/EE380/oneimms.html
Load From Memory

• There’s only one load word instruction:

\[ \text{lw} \ \$rt, \text{imm} (\$rs) \]
\[ \$rt = \text{memory} \left[ \text{imm} + \$rs \right] \]

• It’s sort-of an addiu, but uses the ALU result as the memory address to read
lw $rt,imm($rs)
So, What Does \textit{lw} Need?

- Need MemtoReg mux:

  \[
  \text{MemtoReg} = (\text{ir `OP == `LW);}
  \]
  \[
  \text{assign MemtoRegMux} = (\text{MemtoReg} ? \text{m[ALUresult >> 2]} : \text{ALUresult});
  \]

- Of course, we add a new test case:

  `IPACK(m[12], `LW, 2, 1, 255)
  m[256] = 22;

- Can run it here:

  http://aggregate.org/EE380/onelw.html
Store To Memory

- There's only one store word instruction:
  
  \[ \text{sw } \$rt, \text{imm}(\$rs) \]
  
  \[ \text{memory[imm + } \$rs] = \$rt \]

- It's sort-of an \text{lw}...
sw $rt,imm($rs)
What Changes For sw?

• Very similar to lw, but:
  – Need to route data to memory
  – Unlike every other instruction thus far, sw doesn’t write to a register

```c
if (MemWrite) m[ALUresult >> 2] <= r[ir `RT];
if (RegWrite) r[RegDstMux] <= MemtoRegMux;
```

• Of course, we also add a new test case:

  `IPACK(m[12], `SW, 0, 2, 255)`

• Can run it here:
  [http://aggregate.org/EE380/onesw.html](http://aggregate.org/EE380/onesw.html)
Don’t We Need Control Flow?

• How about a branch equals?

```assembly
beq $rs,$rt,lab
if ($rs == $rt) pc = (pc + 4) + (offset * 4)
where offset = (lab - (pc + 4)) / 4
```

• Of course, offset is really imm... and we shift by 2 rather than multiply by 4
beq $rt, $rs, lab  # offset = (lab - (PC + 4)) >> 2
What Changes For \texttt{beq}?

- Need a shift-by-2 unit and another adder
  
  \begin{verbatim}
  assign Shiftleft2 = (Signextend << 2);
  assign BranchAdd = (PCAdd + Shiftleft2);
  \end{verbatim}

- Need ALU to have a zero flag (use $\texttt{rs} - \texttt{rt}$) and mux to use it
  
  \begin{verbatim}
  assign ALUSrc = ((ir `OP != `RTYPE) && (ir `OP != `BEQ));
  assign zero = (res == 0);
  assign Branch = (ir `OP == `BEQ);
  assign BranchZeroMux = ((Branch & Zero) ? BranchAdd:PCAdd);
  \end{verbatim}

- Can run it here:
  \begin{verbatim}
  http://aggregate.org/EE380/onebeq.html
  \end{verbatim}
# can incrementally add paths for more instructions
# always try to reuse as much hardware as possible
<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>Branch</th>
<th>MemRead</th>
<th>MemtoReg</th>
<th>MemWrite</th>
<th>ALUSrc</th>
<th>RegWrite</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>addiu</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>X</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>1</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>
Basic Pipelining

• Single-cycle control signals move through the pipe along with the data

• Divide single-cycle into equal-delay stages, adding buffers between
  – Ideally, \( n \) stages gives \( nX \) throughput
  – Usually \( <nX \), and \( n \) can’t be huge

• Throughput comes from having useful work in all stages – avoiding bubbles
The single cycle design...
IF: Instruction Fetch  
EX: Execute  
WB: Write Back  
ID: Instruction Decode  
MEM: Memory Access  
what is this?
• Add buffers between pipe stages...
Pipeline Throughput

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu $t0,$t1,$t2</td>
<td>addu $t0,$t1,$t2</td>
<td>addu $t0,$t1,$t2</td>
<td>addu $t0,$t1,$t2</td>
<td>addu $t0,$t1,$t2</td>
</tr>
<tr>
<td>and $t3,$t4,$t5</td>
<td>and $t3,$t4,$t5</td>
<td>and $t3,$t4,$t5</td>
<td>and $t3,$t4,$t5</td>
<td>and $t3,$t4,$t5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu $t0,$t1,$t2</td>
<td>addu $t0,$t1,$t2</td>
<td>addu $t0,$t1,$t2</td>
<td>addu $t0,$t1,$t2</td>
<td>addu $t0,$t1,$t2</td>
</tr>
<tr>
<td>and $t3,$t4,$t5</td>
<td>and $t3,$t4,$t5</td>
<td>and $t3,$t4,$t5</td>
<td>and $t3,$t4,$t5</td>
<td>and $t3,$t4,$t5</td>
</tr>
<tr>
<td>addiu $t6,$0,1</td>
<td>addiu $t6,$0,1</td>
<td>addiu $t6,$0,1</td>
<td>addiu $t6,$0,1</td>
<td>addiu $t6,$0,1</td>
</tr>
<tr>
<td>lw $t7,0($t8)</td>
<td>lw $t7,0($t8)</td>
<td>lw $t7,0($t8)</td>
<td>lw $t7,0($t8)</td>
<td>lw $t7,0($t8)</td>
</tr>
<tr>
<td>sw $t1,4($t9)</td>
<td>sw $t1,4($t9)</td>
<td>sw $t1,4($t9)</td>
<td>sw $t1,4($t9)</td>
<td>sw $t1,4($t9)</td>
</tr>
<tr>
<td>beq $t1,$t2,lab</td>
<td>beq $t1,$t2,lab</td>
<td>beq $t1,$t2,lab</td>
<td>beq $t1,$t2,lab</td>
<td>beq $t1,$t2,lab</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

...
Pipeline Bubble: \texttt{nop}

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>and $t3, $t4, $t5</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>addiu $t6, $0, 1</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>lw $t7, 0 ($t8)</td>
<td>addiu $t6, $0, 1</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>sw $t1, 4 ($t9)</td>
<td>lw $t7, 0 ($t8)</td>
<td>addiu $t6, $0, 1</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>beq $t1, $t2, lab</td>
<td>sw $t1, 4 ($t9)</td>
<td>lw $t7, 0 ($t8)</td>
<td>addiu $t6, $0, 1</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t9)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t9)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t9)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t9)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t9)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t9)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t9)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t9)</td>
<td>-</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>and $t3, $t4, $t5</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>addiu $t6, $0, 1</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>lw $t7, 0 ($t8)</td>
<td>addiu $t6, $0, 1</td>
<td>addu $t0, $t1, $t2</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>sw $t1, 4 ($t7)</td>
<td>lw $t7, 0 ($t8)</td>
<td>addiu $t6, $0, 1</td>
<td>addu $t0, $t1, $t2</td>
<td>-</td>
</tr>
<tr>
<td>beq $t1, $t2, lab</td>
<td>sw $t1, 4 ($t7)</td>
<td>lw $t7, 0 ($t8)</td>
<td>addiu $t6, $0, 1</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t7)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t7)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t7)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t7)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t7)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t7)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t7)</td>
<td>-</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>sw $t1, 4 ($t7)</td>
<td>-</td>
</tr>
</tbody>
</table>

Note: The pipeline stages are represented as follows:
- **IF**: Instruction Fetch
- **ID**: Instruction Decode
- **EX**: Execute
- **MEM**: Memory Access
- **WB**: Write Back

The pipeline stages are repeated multiple times for each instruction, and the operations are performed in parallel.
## Setting The Stages

<table>
<thead>
<tr>
<th>Instruction</th>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
<th>Circuit delay</th>
</tr>
</thead>
<tbody>
<tr>
<td>addu</td>
<td>250</td>
<td>100</td>
<td>300</td>
<td>–</td>
<td>100</td>
<td>750ps</td>
</tr>
<tr>
<td>and</td>
<td>250</td>
<td>100</td>
<td>100</td>
<td>–</td>
<td>100</td>
<td>550ps</td>
</tr>
<tr>
<td>addiu</td>
<td>250</td>
<td>100</td>
<td>300</td>
<td>–</td>
<td>100</td>
<td>750ps</td>
</tr>
<tr>
<td>lw</td>
<td>250</td>
<td>100</td>
<td>300</td>
<td>250</td>
<td>100</td>
<td>1000ps</td>
</tr>
<tr>
<td>sw</td>
<td>250</td>
<td>100</td>
<td>300</td>
<td>100</td>
<td>100</td>
<td>850ps</td>
</tr>
<tr>
<td>beq</td>
<td>250</td>
<td>100</td>
<td>300</td>
<td>–</td>
<td>–</td>
<td>750ps</td>
</tr>
</tbody>
</table>

- Single-cycle limited by `lw` to **1GHz clock**
- 5-stage pipeline limited by `EX`
  - 300ps latency allows **3.33GHz clock**
  - If added buffers take 100ps, **2.5GHz**
- Multi-cycle might be **5 cycles @2.5GHz**
Why A Bubble?

• **Structural hazards**: can’t do 2 things on one unit of HW – single-cycle cures this!

• **Data dependence**: need results from previous computations – not yet done?

• **Control dependence**
  - Computation of branch target address
  - Conditional branch/jump taken or not?
Dependence Analysis

- **Use or R**: reads the value of a name
- **Def or W**: binds a new value to a name
- **True dep.**: carries a value, $D \rightarrow U$, RAW
  
  \[
  \text{add } \$t0, \$t1, \$t2 \quad \text{or} \quad \$t3, \$t0, \$t4
  \]

- **Anti-dep.**: kills a value, $U \leftarrow D$, WAR
  
  \[
  \text{add } \$t0, \$t1, \$t2 \quad \text{or} \quad \$t1, \$t3, \$t4
  \]

- **Output dep.**: kills a value, $D \rightarrow D$, WAW
  
  \[
  \text{add } \$t0, \$t1, \$t2 \quad \text{or} \quad \$t0, \$t3, \$t4
  \]
When Dependence Matters

• True dependence causes a delay

  add $t0, t1, t2
  or $t3, $t0, $t4  //wait for $t0

• Other types don’t
How To Handle Dependence

- Programmer/assembler pads with NOPs
  - Violates ISA concept (how many NOPs?)
  - too little padding gives wrong answers,
  - too much wastes time
  - Makes program bigger

```
IF   ID   EX   MEM   WB
add $t0,$t1,$t2
nop  nop  nop  nop  ...
add $t0,$t1,$t2
nop  nop  nop  ...
add $t0,$t1,$t2
nop  nop  ...
add $t0,$t1,$t2
nop  ...
...  ...  ...  ...
...  ...  ...  ...
...  ...  ...  ...
...  ...  ...  ...
...  ...  ...  ...
```
How To Handle Dependence

- **Hardware interlock**
  - rs or rt in ID is dest in EX, MEM, WB
  - Detects dep. & stalls until satisfied
  - **nop** is or with side-effects disabled: MemRead, MemWrite, RegWrite, & Branch

<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td>add $t0,$t1,$t2</td>
<td>add $t0,$t1,$t2</td>
<td>...</td>
<td>add $t0,$t1,$t2</td>
<td>...</td>
</tr>
<tr>
<td>or $t3,$t0,$t4</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>or $t3,$t0,$t4</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
</tr>
<tr>
<td>or $t3,$t0,$t4</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>or $t3,$t0,$t4</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
</tr>
<tr>
<td>or $t3,$t0,$t4</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
<td>or $t3,$t0,$t4</td>
<td>...</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
How To Handle Dependence

• **Value forwarding**: use interlock circuitry to find value & forward it to ID stage out
  - ALU ready from EX, so **NO DELAY**!
  - `lw` ready from MEM, so **1 cycle delay**
  - Value still gets stored in register in WB
  - Works great, but adds datapaths…

```plaintext
<table>
<thead>
<tr>
<th>IF</th>
<th>ID</th>
<th>EX</th>
<th>MEM</th>
<th>WB</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>lw $t0,0($t1)</code> or <code>t3,$t0,$t4</code></td>
<td>...</td>
<td><code>lw $t0,0($t1)</code> or <code>t3,$t0,$t4</code></td>
<td>...</td>
<td><code>lw $t0,0($t1)</code> or <code>t3,$t0,$t4</code></td>
</tr>
</tbody>
</table>
| ... | ... | ... | ... | ...
| ... | ... | ... | ... | ...
| ... | ... | ... | ... | ...
| ... | ... | ... | ... | ...
```
Computing The Branch Target

- Branch instructions encode offset, not address
  - Add PC + offset *typically* takes a cycle
  - Hardware interlock & stall waiting
To Take, Or Not To Take?

• When do we know if conditional is taken or not taken?
  - Determined only after EX stage?

• HW interlock very inefficient!
  - Usually determined in a late pipe stage
  - Blocks ALL progress
IF: Instruction Fetch  EX: Execute  WB: Write Back
ID: Instruction Decode  MEM: Memory Access  what is this?
IF: Instruction Fetch  EX: Execute  WB: Write Back
ID: Instruction Decode  MEM: Memory Access
Branch Prediction. Do I Feel Lucky?

Guess wrong and some instructions are gonna die (well, we actually say they’re squashed)

• **Always not taken** – the easiest guess

• Later, we’ll discuss better ways to guess...
Verilog Implementation

• Like you’d expect:
  - Can reuse basic single-cycle design
  - Each stage becomes it’s own `always`
  - Need multiple copies of some signals, one for each stage that uses them

• Not like you’d expect:
  - Some things don’t follow pipe flow
  - Some non-stages should be separate things
• For example, **who updates the PC?**
  - IF sets $PC = PC + 4$
  - ID sets $PC = \text{branch target}$
  - EX, MEM, or WB forces $PC = PC$ because data dependence blocks the pipe

• How to coordinate access to shared data?
  - Multiple readers is fine, with one writer
  - Could use locks, semaphores, etc.
  - Let only one entity update each: **owner** picks value and **is only writer**
Pipelined Verilog Version

- Organized as parallel-executing chunks:
  - IF stage: reads and writes IF_
  - ID stage: reads IF_, writes ID_
  - EX stage: reads ID_, writes EX_
  - MEM stage: reads EX_, writes MEM_
  - WB stage: reads MEM_
  - Are we running?
  - Are mispredicted inst. squashed?
  - Are we blocked by data dependence or forwarding values?
  - Simulation tracing support
Pipelined Verilog Version
IF: Instruction Fetch stage

- Really simple...
  - Memory is \texttt{WORD}, not byte, so address\texttt{>>=2}
  - The only thing setting \texttt{IF_ir} and \texttt{IF_pc}

// IF: Instruction Fetch stage
always @(posedge clk) if (running && !blocked) begin
  IF_ir <= m[(squash ? target : IF_pc) >> 2];
  IF_pc <= (squash ? target : IF_pc) + 4;
end
ID: Instruction Decode

- Decodes the instruction
- Reads from register file \( r[] \)
- Computes \( \text{beq} \) comparison & target

```vhdl
// ID: Instruction Decode stage
always @(posedge clk) if (running && !ID_Bad) begin
    if (blocked) ... else begin
        case (squashed `OP)
            `RTYPE: begin
                case (squashed `FUNCT)
                    `ADDU: begin
                        RegDst=1; Branch=0; MemRead=0;
                        ALUOp=`ALUADD; MemWrite=0; ALUSrc=0;
                        RegWrite=1; Bad=0; end
                ... endcase ... endcase
            ... endcase
        ... ID_s <= s; ID_t <= t; ... ID_ALUOp <= ALUOp; ... end
    end end
```
Ex: Execute stage

- Contains the ALU for integers & addresses

```verilog
// Ex: Execution stage
always @(posedge clk) if (running) begin
  case (ID_ALUOp)
    `ALUAND: alu = ID_s & ID_src;
    `ALUOR:  alu = ID_s | ID_src;
    `ALUADD: alu = ID_s + ID_src;
    `ALUSUB: alu = ID_s - ID_src;
    `ALUSLT: alu = ID_s < ID_src;
    `ALUXOR: alu = ID_s ^ ID_src;
    default: alu = (ID_src << 16);
  endcase
  EX_alu <= alu;
  EX_t <= ID_t;
  EX_rd <= ID_rd;
  EX_MemRead <= ID_MemRead;
  EX_MemWrite <= ID_MemWrite;
  EX_Bad <= ID_Bad;
end
```
MEM: MEMory access

- Does a memory read or write
- Uses EX_MemRead for both read and multiplex v

```// MEM: data MEMory access stage
always @(posedge clk) if (running) begin
  if (EX_MemRead) v = m[EX_alu >> 2]; else v = EX_alu;
  if (EX_MemWrite) m[EX_alu >> 2] <= EX_t;

  MEM_v <= v;
  MEM_rd <= EX_rd;
  MEM_Bad <= EX_Bad;
end```
**WB: Write Back**

- Writes result into register...
  - Register $0$ is read only
  - Not writing? Say we write register 0...

- An instruction isn’t done until it’s here, so this is where halting really happens

```verilog
// WB: register Write Back stage
always @(posedge clk) if (running) begin
  if (MEM_rd) r[MEM_rd] <= MEM_v;
  if (MEM_Bad) halt <= 1;
end
```
Running?

- Enables normal operation of stages
- Not running normally if:
  - Halted
  - There’s a reset in progress; reset writes into stuff it doesn’t own, so we need owners disabled

// Running state?
wire running;
assign running = ((!halt) && (!reset));
Squashed?

- For `beq`, predict not taken, so `IF_pc+4`
- If we were right, no bubble
- If wrong, squash fetched instructions
  - No side-effects to undo yet
  - Convert into `NOP` to prevent future s-e

```plaintext
\`define NOP `OR  // Null OPeration is or $0,$0,$0

// Squash instruction fetched on a mispredicted branch
wire `INST squashed;
assign squashed = (squash ? `NOP : IF_ir);
```
Data Dep. / Forwarding

- Need $rs, $rt unless either is $0
- Desired value could be in EX, MEM, or WB
- Forward priority to EX over MEM over WB
- Can’t forward lw value from EX stage!

// Are we blocked by a dependence on rs or rt?
// Also handles all forwarding
...
assign canfwds = ID_needs && ((EX_deps && !MemRead) || (!EX_deps) && (MEM_deps || WB_deps));
assign canfwdt = ID_needt && ((EX_dept && !MemRead) || (!EX_dept) && (MEM_dept || WB_dept));
assign fwds = (EX_deps ? alu : (MEM_deps ? v : MEM_v));
assign fwdt = (EX_dept ? alu : (MEM_dept ? v : MEM_v));
assign dep = ((ID_needs && (EX_deps || MEM_deps || WB_deps)) || (ID_needt && (EX_dept || MEM_dept || WB_dept)));
assign blocked = (dep && ((ID_needs && !canfwds) || (ID_needt && !canfwdt)));

Simulation Tracing

• More (complex!) parallel-executing stuff:

```vhdl
// State-by-state trace
`ifdef TRACE
always @(posedge clk) if (running) begin
    ...
    $display("IF ir=%x pc=%ld", IF_ir, IF_pc);
    case (IF_ir `OP)
        `RTYPE: begin
            case (IF_ir `FUNCT)
                `ADDU: $display("IF addu $%ld,$%ld,$%ld",
                    IF_ir `RD, IF_ir `RS, IF_ir `RT);
            endcase
        endcase
    if (ID_Bad) $display("ID illegal instruction");
    else $display("ID  s=%ld t=%ld src=%ld rd=%ld MemRead=%b ALUOp=%b MemWrite=%b", ID_s, ID_t,
                    ID_src, ID_rd, ID_MemRead, ID_ALUOp, ID_MemWrite);
    ...
end
`endif
endmodule
```
Pipelined Verilog Version

Color key: initialization
IF squashed ID blocked EX MEM WB debugging
Three Verilog Implementations

- **Multi-cycle** MIPS, multiple CPI:
  
  http://aggregate.org/EE380/multiv.html

- **Single-cycle**, 1 CPI, but slow clock:
  
  http://aggregate.org/EE380/onebeq.html

- **Pipelined**, fast clock, ~1 CPI throughput:
  
  http://aggregate.org/EE380/pipe.html