Complete implementation

MIPS-lite
- arithmetic/logical: add, sub, and, or, slt
- memory access: lw, sw
- branch/jump: beq, j

Combine datapaths for
- instruction fetch (Fig. 5.5)
- R-type operations (Fig. 5.7)
- Load and store (Fig. 5.9)
- Branch (Fig. 5.10)
- Jump (to be added)

Add control signals
Version 1: execute each instruction in 1 clock cycle
Version 2: execute each instruction in multiple (shorter) clock cycles
Combining datapaths: R-type and memory

Version 1: execute instruction in 1 clock cycle
   No datapath resource can be used more than once in a single instruction
   If needed more than once, must be duplicated
   Separate instruction, data memory
Some resources can be shared between different instruction types
   Need to have multiple inputs
   Control signal to select which to use: multiplexor
Arithmetic-logical (R-type) and memory access (load/store) are similar in some ways
Differences
   Second operand input to ALU:
      Register contents for R-type
      Sign-extended immediate value (offset) for load/store
   Value stored in destination register:
      ALU output for R-type
      Data memory value for load
To combine datapaths:
   Select source of second ALU operand
   Select source of data to write to register
Use 2 MUXes with control inputs
   ALUsrc for ALU
   MemtoReg for register write
Fig. 5.11
Combining datapaths: instruction fetch

Can add instruction fetch:

Need separate adder in order to:
- increment PC
- perform ALU operation in same clock cycle

Fig. 5.12
Combining datapaths: branch

Add branch:

Additional MUX uses PCSrc input to select:
- incremented PC or (PC + immediate) from second adder
- output goes to update PC

Need to keep separate adder to compute branch address
Control unit

How to control the datapath?
Need to generate the control signals for MUXes, registers, and memory

Look at instruction formats

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>000000</td>
<td>$rs</td>
<td>$rt</td>
<td>$rd</td>
<td>shamt</td>
<td>function</td>
</tr>
<tr>
<td></td>
<td>[31-26]</td>
<td>[25-21]</td>
<td>[20-16]</td>
<td>[15-11]</td>
<td>[10-6]</td>
<td>[5-0]</td>
</tr>
<tr>
<td>memory</td>
<td>35 or 43</td>
<td>$rs</td>
<td>$rt</td>
<td>address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[31-26]</td>
<td>[25-21]</td>
<td>[20-16]</td>
<td>[15-0]</td>
<td></td>
<td></td>
</tr>
<tr>
<td>branch</td>
<td>4</td>
<td>$rs</td>
<td>$rt</td>
<td>address</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>[31-26]</td>
<td>[25-21]</td>
<td>[20-16]</td>
<td>[15-0]</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Observations:

Opcode is always in upper 6 bits: Op[5-0], function in lower 6 bits: F[5-0]
Use to select operation to perform

Registers to read ($rs, $rt) always in bits 25-21 and 20-16
Use as indexes to register file for read

Base register for load/store always in bits 25-21

Offset always in bits 15-0

Destination register may be in 2 places:
bits 15-11 for R-type
bits 20-16 for load
Use as index to register file to write: need MUX to select
Control unit

New features:

Instruction bit numbers for register numbers, opcode, function
MUX to select destination register

RegDst: selects $rd or $rt to write data

ALU control: uses function code and ALUOp to generate ALU operation selection

What is ALUOp? 2-bit code generated by main control (stay tuned)

Note that the values of RegDst, ALUSrc, and PCSrc are reversed in this diagram.

The version in the current printing of the text is correct.
Control unit: main

Main control unit generates
Write signals for registers and memory
Control input for MUXes
ALUOp input (2 bits)
Multiple levels of control: reduces size of main control unit
Control unit: ALU

ALU operation selection: function depends on type of instruction
  R-type: function determined by function field from instruction
  Load/store: add to compute memory address
  Branch: subtract to compare operands

<table>
<thead>
<tr>
<th>Instruction</th>
<th>ALUOp</th>
<th>Function</th>
<th>ALU control</th>
</tr>
</thead>
<tbody>
<tr>
<td>lw, sw</td>
<td>00</td>
<td>xxxxxxx</td>
<td>010 add</td>
</tr>
<tr>
<td>beq</td>
<td>01</td>
<td>xxxxxxx</td>
<td>110 subtract</td>
</tr>
<tr>
<td>add</td>
<td>10</td>
<td>100000</td>
<td>010 add</td>
</tr>
<tr>
<td>sub</td>
<td>10</td>
<td>100010</td>
<td>110 subtract</td>
</tr>
<tr>
<td>and</td>
<td>10</td>
<td>100100</td>
<td>000 and</td>
</tr>
<tr>
<td>or</td>
<td>10</td>
<td>100101</td>
<td>001 or</td>
</tr>
<tr>
<td>slt</td>
<td>10</td>
<td>101010</td>
<td>111 set on less than</td>
</tr>
</tbody>
</table>

Fig. 5.15 shows complete truth table for generating these ALU control bits
Control unit: main

Main control signals
ALUOp: 2 bits based on op code used as input by ALU control
RegDst: selects from instruction bits 20-16 or 15-11 ($rt or $rd) for destination register to write data
RegWrite: enables writing of destination register
ALUSrc: selects second input of ALU
  0: Read data 2 from register file
  1: sign-extended immediate from instruction
PCSrc: selects input to update PC
  0: PC + 4 from adder
  1: PC + offset from branch target calculation (other adder)
MemRead: data is read from data memory
MemWrite: data is written to data memory
MemtoReg: selects data to send to register file to write
  0: ALU result
  1: data read from memory
Can set all of these based only on opcode, except PCSrc
  Should be set based on beq instruction AND ALU output is 0
Control unit generates Branch signal, which is ANDed with ALU Zero output
Datapath with control

How to set control bits?
Could use Boolean equations or truth tables, but depends only on opcodes

Fig. 5.19
### Table

<table>
<thead>
<tr>
<th>Instruction</th>
<th>RegDst</th>
<th>ALUSrc</th>
<th>MemtoReg</th>
<th>RegWrite</th>
<th>MemRead</th>
<th>MemWrite</th>
<th>Branch</th>
<th>ALUOp1</th>
<th>ALUOp0</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-format</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>lw</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sw</td>
<td>X</td>
<td>1</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>beq</td>
<td>X</td>
<td>0</td>
<td>X</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
</tbody>
</table>

### Figure 5.20

Implementation in gates, using input from opcode bits, in Fig. C.5:

**Op Code bits**

**Control signals**
1. Fetch instruction and increment PC
2. Obtain operands from register file, based on source register numbers
3. Perform ALU operation, using ALU control to select, ALUSrc = 0
4. Select output from ALU using MemtoReg = 0
5. Write back to destination register (RegWrite = 1, RegDst = 1 for $rd)

Note that this entire path is combinational, but the values are generated in the order shown.
Datapath: memory access (load)

1. Fetch instruction and increment PC
2. Obtain base register operand (Read data 1) from register file
3. Perform addition of register value with sign-extended immediate operand in ALU, using ALU control to select operation, ALUSrc = 1 to select immediate
4. Use ALU result as address for data memory
5. Use MemtoReg = 1 to select Read data and write back to destination register
   Controls: RegWrite = 1, RegDst = 0 for $rt

How would this have to change for a store instruction?
1. Fetch instruction and increment PC
2. Obtain base register (Read data 1) and data (Read data 2) from register file
3. Perform addition of register value with sign-extended immediate operand in ALU, using ALU control to select operation, ALUSrc = 1 to select immediate
4. Use ALU result as address for data memory
5. Using MemWrite = 1, write data operand to memory address

Note that MemtoReg and RegDst are don’t cares
1. Fetch instruction and increment PC
2. Read 2 registers from register file for comparison
3. ALU subtracts data values, using ALU control to select operation and ALUSrc = 0
4. Generate branch address: add (PC + 4) to sign-extended offset, shifted left by 2
   why shifted by 2? what control signal values?
5. Use Zero output from ALU (and Branch control) to determine which result to use to update PC
   If equal, use branch address
   else use incremented PC

Note that this is all happening simultaneously in 1 clock cycle (adder separate from ALU)
Datapath: jump

Jump is still missing

Recall format:

<table>
<thead>
<tr>
<th>000010</th>
<th>target</th>
</tr>
</thead>
<tbody>
<tr>
<td>b_{31-26}</td>
<td>b_{25-0}</td>
</tr>
</tbody>
</table>

Compare to branch:

- computes target address differently
  
  \[
  PC \leftarrow PC_{31-28} :: IR_{25-0} :: 00
  \]

- update the PC by using:
  - upper 4 bits of the program counter
  - 26 bits of the target (lower 26 bits of instruction register)
  - two 0's
  (creates a 32-bit address)

unconditional
Datapath: jump

Shift instruction bits 25-0 left 2 bits to create 28 bit value
Combine with bits 31-28 of (PC + 4) to produce 32-bit jump address
Additional MUX uses Jump control to select instruction address
  0: Incremented PC or branch target
  1: Jump address
Datapath: critical path

Why don't we use single-cycle datapath? Too slow!
Clock cycle time must be as long as the longest path: load instruction

5 functional units: PC, instruction memory, register file, ALU, data memory, plus 2 MUXes
Datapath: performance

Single-cycle performance

Suppose operation times are:
- Memory: 2 nanoseconds
- ALU/adders: 2 ns
- Register file (read or write): 1 ns

Assume MUXes, control units, PC access, sign-extend have no delay

How long must the clock cycle be?

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Inst mem</th>
<th>Reg read</th>
<th>ALU op</th>
<th>Data mem</th>
<th>Reg write</th>
<th>Total</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>44%</td>
</tr>
<tr>
<td>load</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>24%</td>
</tr>
<tr>
<td>store</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>12%</td>
</tr>
<tr>
<td>branch</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>18%</td>
</tr>
<tr>
<td>jump</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2%</td>
</tr>
</tbody>
</table>

With single-cycle, clock period must be 8 ns

But, not all instructions are loads! Suppose distribution of instruction types shown.
If we could vary the instruction time, the average would be:

\[(6 \times 44\%) + (8 \times 24\%) + (7 \times 12\%) + (5 \times 18\%) + (2 \times 2\%) = 6.3\]

This is 27% faster!
**Datapath: performance**

Average instruction time could be less if we didn't have to use load instruction to determine cycle time.

But that's not actually the longest instruction time!

What about multiply and divide, or even floating-point operations?

Need separate, slower ALUs.

Assume: 8 ns for floating-point add, 16 ns for floating-point multiply.

<table>
<thead>
<tr>
<th>Instruction type</th>
<th>Inst mem</th>
<th>Reg read</th>
<th>ALU op</th>
<th>Data mem</th>
<th>Reg write</th>
<th>Total</th>
<th>Distribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>R-type</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>6</td>
<td>27%</td>
</tr>
<tr>
<td>load</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>31%</td>
</tr>
<tr>
<td>store</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>7</td>
<td>21%</td>
</tr>
<tr>
<td>branch</td>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>5</td>
<td>5%</td>
</tr>
<tr>
<td>jump</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>2</td>
<td>2%</td>
</tr>
<tr>
<td>FP add</td>
<td>2</td>
<td>1</td>
<td>8</td>
<td>0</td>
<td>1</td>
<td>12</td>
<td>7%</td>
</tr>
<tr>
<td>FP multiply</td>
<td>2</td>
<td>1</td>
<td>16</td>
<td>0</td>
<td>1</td>
<td>20</td>
<td>7%</td>
</tr>
</tbody>
</table>

Average is 8.1 ns, which is only 8.1/20 = 41% of the single-cycle time.

In other words, variable cycle could be about 2.5 times faster!

Variable clock cycle is impractical, but we can break up the instructions into shorter cycles, then only use the parts needed for any given instruction.