Computer organization

- Computer design – an application of digital logic design procedures
- Computer = processing unit + memory system
- Processing unit = control + datapath
- Control = finite state machine
  - inputs = machine instruction, datapath conditions
  - outputs = register transfer control signals, ALU operation codes
  - instruction interpretation = instruction fetch, decode, execute
- Datapath = functional units + registers
  - functional units = ALU, multipliers, dividers, etc.
  - registers = program counter, shifters, storage registers

Structure of a computer

- Block diagram view

![Diagram of computer structure](image-url)
Registers

- Selectively loaded – EN or LD input
- Output enable – OE input
- Multiple registers – group 4 or 8 in parallel

OE asserted causes FF state to be connected to output pins; otherwise they are left unconnected (high impedance)

LD asserted during a lo-to-hi clock transition loads new data into FFs

Register transfer

- Point-to-point connection
  - dedicated wires
  - muxes on inputs of each register

- Common input from multiplexer (input bus)
  - load enables for each register
  - control signals for multiplexer

- Common bus with output enables (input/output bus)
  - output enables and load enables for each register
Register files

■ Collections of registers in one package
  ■ two-dimensional array of FFs
  ■ address used as index to a particular word
  ■ can have separate read and write addresses so can do both at same time
■ 4 by 4 register file
  ■ 16 D-FFs
  ■ organized as four words of four bits each
  ■ write-enable (load)
  ■ read-enable (output enable)

Memories

■ Larger collections of storage elements
  ■ implemented not as FFs but as much more efficient latches
  ■ high-density memories use 1 to 5 switches (transitors) per memory bit
■ Static RAM – 1024 words each 4 bits wide
  ■ once written, memory holds as long as there is power applied
    ■ not true for denser dynamic RAM – lose power, lose memory
  ■ address lines to select word (10 lines for 1024 words)
  ■ read enable
    ■ same as output enable
    ■ often called chip select (CS)
    ■ permits connection of many chips into larger array
      (tie multiple chips IO pins together)
  ■ write enable (same as load)
  ■ bi-directional data lines
    ■ output when reading, input when writing
Instruction sequencing

- Example – an instruction to add the contents of two registers (Rx and Ry) and place result in a third register (Rz)
- Step 1: get ADD instruction from memory into instruction register (IR)
- Step 2: decode instruction
  - instruction in IR has “operation code” to identify it as an ADD instruction
  - register indices used to generate output enables for registers Rx and Ry
  - register index used to generate load signal for register Rz
- Step 3: execute instruction
  - enable Rx and Ry output and direct to ALU (possibly through busses/muxes)
  - set ALU to perform ADD operation
  - direct result (through busses/muxes) to Rz so that it can be loaded into register

Instruction types

- Data manipulation
  - add, subtract
  - increment, decrement
  - multiply
  - shift, rotate
  - immediate operands
- Data staging
  - load/store data to/from memory
  - register-to-register move
- Control
  - conditional/unconditional branches in program flow
  - subroutine call and return
Elements of the control unit (aka instruction unit)

- Standard FSM elements
  - state register
  - next-state logic
  - output logic (data-path/control signaling)
  - Moore or synchronous Mealy machine (to avoid loops unbroken by FF)
- Plus additional “control” registers
  - instruction register (IR)
  - program counter (PC)
- Inputs/outputs
  - outputs control elements of data path
  - inputs from data path used to alter flow of program (e.g., test if zero)

Instruction execution

- Control state diagram (for each diagram)
  - reset
  - fetch instruction
  - decode
  - execute
- Instructions partitioned into three classes
  - branch
  - load/store
  - register-to-register
- Different sequence through diagram for each instruction type (may need more than one state)
Data path (hierarchy)

- Arithmetic circuits constructed in hierarchical and modular fashion
  - each bit in datapath is functionally identical
  - 4-bit, 8-bit, 16-bit, 32-bit datapaths
  - may include carry-lookahead or carry-select capability

Data path (ALU)

- ALU block diagram
  - input: data on which to operate and operation to perform
  - output: result of operation and status information
Data path (ALU + registers)

- Accumulator (common register construct)
  - special register
  - one of the inputs to ALU
  - output of ALU always stored back in accumulator
- One-address instructions
  - only need operation and address of one operand
  - other operand and destination is accumulator register
  - $AC \leftarrow AC <\text{op}> \text{Mem}[\text{addr}]$
  - "single address instructions" (AC implicit operand)
- Multiple registers
  - part of instruction used to choose register operands

Data path (bit-slice)

- Bit-slice concept – replicate to build n-bit wide datapaths
Instruction path

- Program counter (PC)
  - keeps track of program execution
  - address of next instruction to read from memory
  - may have “auto-increment” feature or use ALU to “add 1”
- Instruction register (IR)
  - current instruction
  - includes ALU operation and address(es) of operand(s)
  - also holds target of jump instruction if branch instruction
  - immediate operands – value represented explicitly in instruction
- Relationship to data path
  - PC may be incremented through ALU
  - contents of IR may also be required as input to ALU – immediate operands

Data path (memory interface)

- Memory
  - separate data and instruction memory (Harvard architecture)
    - two address busses, two data busses
  - single combined memory (Princeton architecture)
    - single address bus, single data bus
- Separate memory
  - ALU output -> data memory input
  - instruction register -> data memory address
  - data memory output -> input to registers
  - program counter -> instruction memory address
  - instruction memory output -> instruction register
- Single memory
  - ALU output -> memory input
  - PC or IR -> address
  - memory output -> instruction or data registers
**Block diagram of processor (Harvard)**

- Register transfer view of Harvard architecture
  - black arrows represent data-flow between registers
  - blue arrows other are control signals from control FSM (also load control for each register, not shown)
  - 2 MARs (PC and IR)
    - MAR = memory address register
  - 3 MBRs (AC, REG and IR)
    - MBR = memory buffer register

**Block diagram of processor (Princeton)**

- Register transfer view of Princeton architecture
  - black arrows represent data-flow between registers
  - blue arrows other are control signals from control FSM (also load control and output enable for each register, not shown)
  - 2 MARs (PC and IR) multiplexed (3-state)
    - MAR = memory address register
  - 3 MBRs (AC, REG and IR)
    - MBR = memory buffer register
A simplified processor data-path and memory

- Modeled after MIPS R2000
  - Used in 378 text by Patterson & Hennessy
  - Princeton architecture – shared data/instruction memory
  - 32-bit machine
  - 32 register file
  - PC incremented through ALU
  - Multi-cycle instructions in our implementation
    - single-cycle for real R2000, you’ll see that in 378
  - Only a subset of the instructions are implemented
  - Synchronous Mealy (Moore) controller

Processor instructions

- Three principal types (32 bits in each instruction)

<table>
<thead>
<tr>
<th>type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shft</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>R(register)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>I(immediate)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J(jump)</td>
<td>6</td>
<td></td>
<td></td>
<td>26</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- The instructions we will implement (only a small subset)

<table>
<thead>
<tr>
<th>R</th>
<th>type</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
<th>rd</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>add</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td></td>
<td>sub</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td></td>
<td>and</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td></td>
<td>or</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td></td>
<td>blt</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>42</td>
</tr>
<tr>
<td>I</td>
<td>lw</td>
<td>35</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td>rt</td>
<td>= rs + rt</td>
</tr>
<tr>
<td></td>
<td>sw</td>
<td>43</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td>rt</td>
<td>mem[rs + offset] = rt</td>
</tr>
<tr>
<td></td>
<td>beq</td>
<td>4</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td>pc</td>
<td>= pc + offset, if (rs == rt)</td>
</tr>
<tr>
<td></td>
<td>addi</td>
<td>8</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td>rt</td>
<td>= rs + offset</td>
</tr>
<tr>
<td>J</td>
<td>halt</td>
<td>2</td>
<td></td>
<td></td>
<td>target address</td>
<td>pc</td>
<td>= target address</td>
</tr>
<tr>
<td></td>
<td>63</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>stop execution until reset</td>
<td></td>
</tr>
</tbody>
</table>
Our R2000 implementation

module Memory(address, write, read, data);
    input [31:0] address;
    input write, read;
    inout [31:0] data;
    reg [31:0] memory[0:255];
    wire delayed_write;
    assign #10 delayed_write = write;
    always @(posedge delayed_write) begin
        memory[address[7:0]] = data;
    end
    assign data = read ? memory[address[7:0]] : 32'hzzzzzzzz;
endmodule
Program (compute $n^{th}$ Fibonacci number)

```plaintext
r0 = mem[254];
rl = 0;
r2 = 1;

if (r0 > 0) goto entry; else goto exit2;
loop:
rl = rl + r2;  
r0 = r0 - 1;
entry:
if (r0 == 0) goto exit1;  
r2 = rl + r1;
r0 = r0 - 1;
if (r0 == 0) goto exit2; else goto loop;
exit1:
mem[255] = r2;
HALT
```

Memory – initial contents (test fixture)

```plaintext
parameter shftX = 5'hxx;
parameter r0 = 5'h00;
parameter rl = 5'h01;
parameter r2 = 5'h02;
parameter r3 = 5'h03;
parameter rz = 5'h1f;

initial begin
memory[8'h001] = (ADDI, rl, rl, 16'h0000);
// r0 = 0
memory[8'h002] = (ADDI, rl, rl, 16'h0001);
// r2 = 1
memory[8'h003] = (LW, rl, r0, 16'h000e);
// r0 = mem[254]
memory[8'h004] = (BEQ, r3, rl, 16'h0009);
// if (r3 == r0) goto exit2
memory[8'h005] = (BEQ, rl, rl, 16'h0002);
// if (rl == rl) goto entry /* goto entry
memory[8'h006] = (ALU, rl, rl, r1, shftX, ADD);
// loop: rl = rl + r2
memory[8'h007] = (ALU, rl, r0, 16'h00f6);
// r0 = r0 + 1
memory[8'h008] = (BEQ, rl, rl, 16'h0004);
// entry: if (rl == rl) goto exit1
memory[8'h009] = (ALU, rl, rl, rl, shftX, ADD);
// r2 = rl + r2
memory[8'h00a] = (ADDI, rl, r0, 16'h00f7);
// r0 = r0 + 1
memory[8'h00b] = (BEQ, rl, rl, 16'h0002);
// if (rl == rl) goto exit2
memory[8'h00c] = (J, 26'h00000006);
// goto loop
memory[8'h00d] = (ALU, rl, rl, rl, shftX, OR);
// exit1: r2 = rl + r2 */ r2 = r2
memory[8'h00e] = (LRW, rl, r2, 16'h00f5);
// exit2: mem[255] = rl
memory[8'h00f] = (HALT, 26'hxxxxxxx);
// halt
memory[8'hfe] = 32'h00000004;
// this is the input N
end
```
ALU

module ALU(RegA, PC, Inst, RegB, op, srcA, srcB, ALUout, zero, neg);
  input [31:0] RegA;
  input [31:0] PC;
  input [31:0] Inst;
  input [31:0] RegB;
  input [31:0] op;
  input srcA;
  input srcB;
  output [31:0] ALUout;
  output zero, neg;
  wire [31:0] A;
  reg [31:0] B;
  reg [31:0] result;
  reg zero;
  reg neg;
  assign A = (srcA ? PC : RegA);
  always @(A or B or op) begin
    case (op)
      6'b000001: result = A + B;
      6'b000010: result = A - B;
      6'b000100: result = A & B;
      6'b001000: result = A | B;
      6'b010000: result = A;
      6'b100000: result = B;
      default: result = 32'hxxxxxxxx;
    endcase
    zero = (result == 32'h00000000);
    neg = result[31];
  end
  assign ALUout = result;
endmodule

Registers and 3-state drivers

module Reg32_LD(D, LD, Q, clk);
  input [31:0] D;
  input LD;
  output [31:0] Q;
  input clk;
  reg [31:0] Q;
  always @ (posedge clk) begin
    if (LD) Q = D;
  end
endmodule

module Tri32(I, OE, O);
  input [31:0] I;
  input OE;
  output [31:0] O;
  assign O = (OE) ? I : 32'hzzzzzzzz;
endmodule
PC – a special register

```verilog
module PC(ALUout, Inst, reset, PCsel, PCld, clk, PC);
  input [31:0] ALUout;
  input [31:0] Inst;
  input reset, PCsel, PCld, clk;
  output [31:0] PC;
  reg [31:0] PC;
  wire [31:0] src;
  assign src = PCsel ? ALUout : {6'b000000, Inst[25:0]};
  always @(posedge clk) begin
    if (reset) PC = 32'h00000000;
    else if (PCld) PC = src;
  end
endmodule
```

Register file

```verilog
module RegFile(MBR, ALUout, Inst, regWrite, wrDataSel, wrRegSel, RegA, RegB, clk);
  input [31:0] MBR;
  input [31:0] ALUout;
  input [31:0] Inst;
  input regWrite, wrDataSel, wrRegSel;
  output [31:0] RegA;
  output [31:0] RegB;
  input clk;
  wire [4:0] rs, rt, rd, wrReg;
  wire [31:0] wrData;
  reg [31:0] RegFile[0:31];
  reg [31:0] RegA, RegB;
  initial begin
    RegFile[31] = 0;
  end
  assign rs = Inst[25:21];
  assign rt = Inst[20:16];
  assign rd = Inst[15:11];
  assign wrReg = wrRegSel ? rd : rt;
  assign wrData = wrDataSel ? MBR : ALUout;
  always @(posedge clk) begin
    RegA = RegFile[rs];
    RegB = RegFile[rt];
    if (regWrite && (wrReg != 31)) begin
      RegFile[wrReg] = wrData;
    end
  end
endmodule
```
Tracing an instruction's execution

- Instruction: \( r3 = r1 + r2 \)
  - \( R \)
    - \( rs=r1 \)
    - \( rt=r2 \)
    - \( rd=r3 \)
    - \( shift=X \)
    - \( funct=32 \)

- 1. instruction fetch
  - move instruction address from PC to memory address bus
  - assert memory read
  - move data from memory data bus into IR
  - configure ALU to add 1 to PC
  - configure PC to store new value from ALUout

- 2. instruction decode
  - op-code bits of IR are input to control FSM
  - rest of IR bits encode the operand addresses (rs and rt) – these go to register file

- 3. instruction execute
  - set up ALU inputs
  - configure ALU to perform ADD operation
  - configure register file to store ALU result (rd)

Tracing an instruction's execution (cont’d)
Step 1: \( IR \leftarrow \text{mem}[PC] \);
Tracing an instruction's execution (cont’d)
Step 1: IR ← mem[PC];

Step 1: IR ← mem[PC]; PC ← PC + 1;
Tracing an instruction's execution (cont’d)

Step 1: IR ← mem[PC]; PC ← PC + 1;

- Control signals
  - PCmaEN = 1;
  - mr = 1;
  - IRld = 1;
  - ALUmaEN = 0;
  - mw = 0;
  - RegBmdEN = 0;
  - srcA = "PC" = 1;
  - srcB = "1" = 2'b11;
  - op = "+" = 6'b000001;
  - PCld = 1;
  - PCsel = 1;

- But, also . . .
  - regWrite = 0;
  - wrDataSel = X;
  - wrRegSel = X;
  - MBRld = X;

- At end of cycle, IR is loaded with instruction that will be seen by controller
  - But, control signals for instruction can’t be output until next cycle
  - One cycle just for signals to propagate (Step 2)

Step 2: RegA ← regfile[rs]; RegB ← regfile[rt];
Tracing an instruction's execution (cont’d)
Step 2: \( \text{RegA} \leftarrow \text{regfile}[rs]; \text{RegB} \leftarrow \text{regfile}[rt]; \)

- **Control signals**
  - \( \text{PCmaEN} = 0; \)
  - \( \text{mr} = X; \)
  - \( \text{IRId} = 0; \)
  - \( \text{ALUmaEN} = 0; \)
  - \( \text{mw} = 0; \)
  - \( \text{RegBmdEN} = 0; \)
  - \( \text{regWrite} = 0; \)
  - \( \text{PCId} = 0; \)
  - \( \text{PCsel} = X; \)

- **But, also . . .**
  - \( \text{srcA} = X; \)
  - \( \text{srcB} = 2'bX; \)
  - \( \text{op} = 6'bXXXXXX; \)
  - \( \text{wrDataSel} = X; \)
  - \( \text{wrRegSel} = X; \)
  - \( \text{MBRId} = X; \)
Tracing an instruction's execution (cont’d)
Step 3: Regfile[rd] ← RegA + RegB;
Tracing an instruction's execution (cont’d)
Step 3: Regfile[rd] ← RegA + RegB;

- Control signals
  - PCmaEN = 0;
  - mr = X;
  - IRId = 0;
  - AL UmaEN = 0;
  - mw = 0;
  - RegBmdEN = 0;
  - srcA = “A” = 0;
  - srcB = “B” = 2'b00;
  - op = “+” = 6'b000001;
  - regWrite = 1;
  - wrDataSel = “ALU” = 0;
  - wrRegSel = “rd” = 1;

- But, also . . .
  - PCld = 0;
  - PCsel = X;
  - MBRId = X;

Register-transfer-level description

- Control
  - transfer data between registers by asserting appropriate control signals

- Register transfer notation - work from register to register
  - instruction fetch:
    - mabus ← PC;  — move PC to memory address bus (PCmaEN, AL UmaEN)
    - memory read;  — assert memory read signal (mr)
    - IR ← memory;  — load IR from memory data bus (IRId)
    - op ← add  — send PC into A input, 1 into B input, add (PC + 1)
      (srcA, srcB[1:0], op)
    - PC ← ALU  — load result of incrementing in ALU into PC (PCld, PCsel)
  - instruction decode:
    - IR to controller
    - values of A and B read from register file (rs, rt)
  - instruction execution:
    - op ← add  — send regA into A input, regB into B input, add (A + B)
      (srcA, srcB[1:0], op)
    - rd ← ALU  — store result of add into destination register
      (regWrite, wrDataSel, wrRegSel)
Register-transfer-level description (cont’d)

- How many states are needed to accomplish these transfers?
  - data dependencies (where do values that are needed come from?)
  - resource conflicts (ALU, buses, etc.)
- In our case, it takes three cycles
  - one for each step
  - all operations within a cycle occur between rising edges of the clock
- How do we set all of the control signals to be output by the state machine?
  - depends on the type of machine (Mealy, Moore, synchronous Mealy)

Review of FSM timing

```
step 1
```

```
fetch
IR ← mem[PC];
```

```
step 2
```

```
declare
PC ← PC + 1;
A ← rs
B ← rt
```

```
step 3
```

```
execute
rd ← A + B
```

to configure the data-path to do this here, when do we set the control signals?
FSM controller for CPU (skeletal Moore FSM)

- First pass at deriving the state diagram (Moore machine)
  - these will be further refined into sub-states

```
reset
instruction fetch
```

```
instruction decode
LW
SW
ADD
```

```
instruction execution
```

FSM controller for CPU (reset and inst. fetch)

- Assume Moore machine
  - outputs associated with states rather than arcs
- Reset state and instruction fetch sequence
- On reset (go to Fetch state)
  - start fetching instructions
  - PC will set itself to zero

```
mabus ← PC;
memory read;
IR ← memory data bus;
PC ← PC + 1;
```

```
reset
Fetch
instruction fetch
```
FSM controller for CPU (decode)

- Operation decode state
  - next state branch based on operation code in instruction
  - read two operands out of register file
    - what if the instruction doesn’t have two operands?

![Diagram showing state transitions in the FSM controller for CPU (decode)]

FSM controller for CPU (instruction execution)

- For add instruction
  - configure ALU and store result in register
    
    \[ \text{rd} \leftarrow A + B \]
  - other instructions may require multiple cycles
FSM controller for CPU (add instruction)

- Putting it all together and closing the loop
  - the famous instruction fetch decode execute cycle

FSM controller for CPU

- Now we need to repeat this for all the instructions of our processor
  - fetch and decode states stay the same
  - different execution states for each instruction
    - some may require multiple states if available register transfer paths require sequencing of steps
Tracing an instruction's execution (LW)
Step 1: IR ← mem[PC]; PC ← PC + 1;

Tracing an instruction's execution (LW cont'd)
Step 2: Instruction propagates through controller
Tracing an instruction's execution (LW cont’d)

Step 3: \( \text{ALUoutReg} \leftarrow \text{regfile[rs]} + \text{offset}; \)

Step 4: \( \text{MBR} \leftarrow \text{mem[ALUoutReg]}; \)
Tracing an instruction's execution (LW cont’d)

Step 5: \( \text{regfile[rt]} \leftarrow \text{MBR}; \)

Controller signals for all cycles (LW cont’d)

- Control signals for:
  - \( \text{PCmaEN} = 1 \)
  - \( \text{ALUmaEN} = 0 \)
  - \( \text{RegBmdEN} = 0 \)
  - \( \text{mr} = 1 \)
  - \( \text{mw} = 0 \)
  - \( \text{IRld} = 1 \)
  - \( \text{MBRld} = X \)
  - \( \text{srcA} = \text{"PC"} \)
  - \( \text{srcB} = \text{"1"} \)
  - \( \text{op} = + \)
  - \( \text{regWrite} = 0 \)
  - \( \text{wrDataSel} = X \)
  - \( \text{wrRegSel} = X \)
  - \( \text{PCld} = 1 \)
  - \( \text{PCsel} = 1 \)
FSM controller (complete state diagram)

Controller

module Controller(Inst, neg, zero, reset, clk, srcA, srcB, op, mr, mw, PCmaEN, ALUmaEN, RegBmdEN, MBRld, IRld, regWrite, wrDataSel, wrRespSel, PCsel, PCld);

input [31:0] Inst;
input neg, zero, reset, clk;
output srcA;
output [1:0] srcB;
output [5:0] op;
output MBRld, IRld, regWrite;
output wrDataSel, wrRespSel, PCsel, PCld;
output mr, mw, PCmaEN, ALUmaEN, RegBmdEN;

reg srcA;
reg MBRld, IRld, regWrite;
reg wrDataSel, wrRespSel, PCsel, PCld;
reg mr, mw, PCmaEN, ALUmaEN, RegBmdEN;

reg [5:0] op;
reg [1:0] srcB;

reg [2:0] state;

wire [5:0] instOp;
wire [5:0] instSubOp;

Controller
Controller

```verilog
class Controller:

    assign instOp = Inst[31:26];
    assign instSubOp = Inst[5:0];

    always @(posedge clk) begin
        state = fetch; end
    else begin
        casex({state, instOp, instSubOp})
        {fetch, DONTCARE, DONTCARE}: state = decode; // fetch cycle
        {decode, DONTCARE, DONTCARE}: state = execute1; // decode cycle
        {execute1, ALU, ADD}: state = fetch; // execute cycle for ALU-ADD
        {execute1, ALU, SLT}: state = (neg ? execute2 : execute3); // 1st execute cycle for ALU-SLT,
        // branch depending on comparison
        {execute2, ALU, SLT}: state = fetch; // 2nd execute cycle for ALU-SLT when rs < rt
        {execute3, ALU, SLT}: state = fetch; // 2nd execute cycle for ALU-SLT when rs >= rt
        {execute1, LW, DONTCARE}: state = execute2; // 1st execute cycle for LW
        {execute2, LW, DONTCARE}: state = execute3; // 2nd execute cycle for LW
        {execute3, LW, DONTCARE}: state = fetch; // 3rd execute cycle for LW
        {execute1, SW, DONTCARE}: state = execute2; // 1st execute cycle for SW
        {execute2, SW, DONTCARE}: state = execute3; // 2nd execute cycle for SW
        {execute3, SW, DONTCARE}: state = fetch; // 3rd execute cycle for SW
        default: state = BADSTATE; // should never get here
    endcase
end
```

```verilog
Controller

always @(state) begin
    // Set defaults that may be overwritten in case statement, just to be safe
    IRld = 0; MBRld = 0; PCld = 0; regWrite = 0;
    mr = 0; mw = 0; ALUoEN = 0; PCen = 0; Regen = 0;
    casex({state, instOp, instSubOp})
    {fetch, DONTCARE, DONTCARE}: begin
        // fetch the instruction and load it into instruction register
        PCen = 1;
        mr = 1;
        IRld = 1;
        // increment PC
        srcA = srcAPC;
        srcB = srcBone;
        op = aluAdd;
        PCsel = pcSelALU;
        PCld = 1;
        end
    {decode, DONTCARE, DONTCARE}: begin
        // propagate signals into controller, nothing to do
        end
    {execute1, ALU, DONTCARE}: begin
        srcA = srcAreg;
        srcB = srcBreg;
        case (instSubOp)
            ADD: op = aluAdd;
            SUB: op = aluSub;
            OR: op = aluOr;
            SLT: op = aluSub;
            endcase
        wrSegSel = regSD;
        wcDataSel = regAD;
        regWrite = 1;
        end
```
Controller

{execute2, ALU, SLT}: begin
    // rs < rt, load a one into rd
    srcB = srcBone;
    op = aluPassB;
    wrSegsel1 = regALU;
    wrDest = regD;
    regWrite = 1;
    end

{execute3, ALU, SLT}: begin
    // rs > rt, load a zero into rd
    srcB = srcBzero;
    op = aluPassB;
    wrSegsel1 = regALU;
    wrDest = regD;
    regWrite = 1;
    end

{execute1, LW, DONTCARE}: begin
    // compute address by adding rs + offset
    // FILL IN OTHER STATES AND CONTROL SIGNALS FOR FIRST LW EXECUTE STATE
    end

{execute2, LW, DONTCARE}: begin
    // read from memory into MBR
    // FILL IN OTHER STATES AND CONTROL SIGNALS FOR SECOND LW EXECUTE STATE
    end

{execute3, LW, DONTCARE}: begin
    // write MBR into register file’s rt
    // FILL IN OTHER STATES AND CONTROL SIGNALS FOR THIRD LW EXECUTE STATE
    end

{execute1, SW, DONTCARE}: begin
    // compute address by adding rs + offset
    // FILL IN OTHER STATES AND CONTROL SIGNALS FOR FIRST SW EXECUTE STATE
    end

// FILL IN OTHER STATES AND CONTROL SIGNALS FOR SW, BEQ, ADDI, J, and HALT
endcase
end
endmodule

Simulation of the processor

<table>
<thead>
<tr>
<th>Name</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>55</th>
<th>60</th>
<th>65</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rst</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_char</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_addr</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_data</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_char</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_addr</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_data</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_pc</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_csr</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regALU</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regRD</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regD</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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Name</th>
<th>0</th>
<th>5</th>
<th>10</th>
<th>15</th>
<th>20</th>
<th>25</th>
<th>30</th>
<th>35</th>
<th>40</th>
<th>45</th>
<th>50</th>
<th>55</th>
<th>60</th>
<th>65</th>
<th>70</th>
</tr>
</thead>
<tbody>
<tr>
<td>clk</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>rst</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_char</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_addr</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>s_data</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_char</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_addr</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_data</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_pc</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>w_csr</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ALU</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regALU</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regRD</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></td>
<td></td>
<td></td>
</tr>
<tr>
<td>regD</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></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Estimating performance

- Recall basic constraint equations:
  - $T_{\text{period}} > T_{\text{FFprop}} + T_{\text{delay}} + T_{\text{setup}}$
  - $T_{\text{prop}} > T_{\text{hold}}$ (this is usually designed into the FFs and is not our concern)
- Clock period is maximum of $T_{\text{period}}$ along all possible paths in the circuit between flip-flops
  - Clock period = 1/frequency = max($T_{\text{period}}$) over all paths
  - Assuming all FFs are the same:
    \[
    \max (T_{\text{period}}) = T_{\text{FFprop}} + \max(T_{\text{delay}}) + T_{\text{setup}}
    \]

Paths between FFs during “fetch” and “decode”

$T_{\text{delay}} = T_{\text{3state}} + T_{\text{memoryread}} + T_{\text{wires}}$

Assume $T_{\text{3state}}$ is small and can be ignored.

Note: this is NOT TRUE in modern chip design.
$T_{\text{delay}} = T_{\text{Amux}} + T_{\text{ALU}} + T_{\text{PCmux}}$

Paths between FFs during “fetch” and “decode”
Estimating performance for “fetch” and “decode” cycles

- Max(T_{delay}) = Max of the paths on previous four slides
  - T_{3state} + T_{memoryread}
  - T_{Amux} + T_{ALU} + T_{PCmux}
  - T_{RegFileRead}
  - T_{controller}

- Which is likely to be largest?
  - T_{3state}, T_{Amux} and T_{PCmux} are likely to be small
  - T_{RegFileRead} is larger (32 register memory – large tri-state mux)
  - T_{ALU} is probably larger as it includes a 32-bit carry (lookahead?)
  - T_{memoryread} is an even larger array (typically an important factor)
  - T_{controller} is the wild card (depends on complexity of logic in FSM)
Estimating performance for “execute” cycles

- Max(Tdelay) = Max of previous as well as
  - $T_{\text{3state}} + T_{\text{memorywrite}}$
  - $T_{\text{Busx}} + T_{\text{ALU}} + T_{\text{controller}}$

- Now $T_{\text{ALU}}$ and $T_{\text{controller}}$ are added together
  - These are two of our potentially largest delays
  - Adding them together will almost surely be the maximum
  - How could this path be broken up so that we separate the ALU and controller’s delays?

Other factors in estimating performance

- Off-chip communication is much slower than on-chip
  - $T_{\text{wires}}$ can’t always be ignored
  - Try to keep communicating elements on one chip
  - Separate onto separate chips at clock boundaries

- Add registers to data-path to separate long propagation delays into smaller pieces
  - Adds more cycles to operations
  - But each cycle is smaller
  - Which is better?
    - more numerous cycles of simple and fast operations
    - fewer cycles of complex and slow operations

- This is what computer architecture is about – see CSE 378