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 showing the structure of a computer with labeled blocks for Processor, Memory System, Control, Data Path, and central processing unit (CPU).]
 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
  - load enables for each register
  - control signals for multiplexer

- Common bus with output enables
  - 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 forever (not true for denser dynamic RAM)
  - address lines to select word (10 lines for 1024 words)
  - read enable
    - same as output enable
    - often called chip select
    - permits connection of many
      chips into larger array
  - write enable (same as load enable)
  - 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 the ADD instruction from memory into an instruction register

Step 2: decode instruction
- instruction in IR has the code of 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
- setup ALU to perform ADD operation
- direct result 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 (datapath/control signalling)
  - 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 (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
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

Data path (ALU)

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

- Accumulator
  - special register
  - one of the inputs to ALU
  - output of ALU stored back in accumulator
- One-address instructions
  - operation and address of one operand
  - other operand and destination is accumulator register
  - AC ← AC op Mem[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
- **Instruction register (IR)**
  - current instruction
  - includes ALU operation and address of operand
  - also holds target of jump instruction
  - immediate operands
- **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 goes to data memory input
  - register input from data memory output
  - data memory address from instruction register
  - instruction register from instruction memory output
  - instruction memory address from program counter
- **Single memory**
  - address from PC or IR
  - memory output to instruction and data registers
  - memory input from ALU output
Block diagram of processor (Harvard)

- Register transfer view of Harvard architecture
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - two MARs (PC and IR)
  - two MBRs (REG and IR)
  - load control for each register

Block diagram of processor (Princeton)

- Register transfer view of Princeton architecture
  - which register outputs are connected to which register inputs
  - arrows represent data-flow, other are control signals from control FSM
  - MAR may be a simple multiplexer rather than separate register (impl. using 3-state)
  - two MBRs (REG and IR)
  - load control for each register
A simplified processor data-path and memory

- Modeled after MIPS R2000
  - used as main example in 378 textbook 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
  - Only a subset of the instructions are implemented
  - Synchronous Mealy or Moore controller

---

Our R2000 implementation
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(egister)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>6</td>
<td></td>
</tr>
<tr>
<td>I(mmediate)</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>J(ump)</td>
<td>6</td>
<td>26</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>R</th>
<th>op</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shft</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>32</td>
</tr>
<tr>
<td>sub</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>34</td>
</tr>
<tr>
<td>and</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>36</td>
</tr>
<tr>
<td>or</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>37</td>
</tr>
<tr>
<td>silt</td>
<td>0</td>
<td>rs</td>
<td>rt</td>
<td>rd</td>
<td>0</td>
<td>42</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>I</th>
<th>lw</th>
<th>rs</th>
<th>rt</th>
<th>offset</th>
<th>rt = mem[rs + offset]</th>
</tr>
</thead>
<tbody>
<tr>
<td>sw</td>
<td>43</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td>mem[rs + offset] = rt</td>
</tr>
<tr>
<td>beq</td>
<td>4</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td>pc = pc + offset, if (rs == rt)</td>
</tr>
<tr>
<td>addi</td>
<td>8</td>
<td>rs</td>
<td>rt</td>
<td>offset</td>
<td>rt = rs + offset</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>J</th>
<th>i</th>
<th>target address</th>
<th>pc = target address</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>2</td>
<td>-</td>
<td>stop execution until reset</td>
</tr>
</tbody>
</table>

Our R2000 implementation
Tracing an instruction's execution

- Instruction: \( r_3 = r_1 + r_2 \)

<table>
<thead>
<tr>
<th>R</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shft</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>X</td>
<td>32</td>
</tr>
</tbody>
</table>

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 ALU out

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

Tracing an instruction's execution (cont’d)

- Instruction: \( r_3 = r_1 + r_2 \)

<table>
<thead>
<tr>
<th>R</th>
<th>rs</th>
<th>rt</th>
<th>rd</th>
<th>shft</th>
<th>funct</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>X</td>
<td>32</td>
</tr>
</tbody>
</table>

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 ← 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)
Tracing an instruction's execution (cont’d)

Step 2: RegA ← regfile[rs]; RegB ← regfile[rt];

- Control signals
  - PCmaEN = 0;
  - mr = X;
  - IRId = 0;
  - ALUmaEN = 0;
  - mw = 0;
  - RegBmdEN = 0;
  - regWrite = 0;
  - PCld = 0;
  - PCsel = X;

- But, also...
  - srcA = X;
  - srcB = 2'bX;
  - op = 6'bXXXXXX;
  - wrDataSel = X;
  - wrRegSel = X;
  - MBRld = X;
Tracing an instruction's execution (cont’d)

Step 3: \( \text{Regfile}[rd] \leftarrow \text{RegA} + \text{RegB}; \)

- **Control signals**
  - \( \text{PCmaEN} = 0; \)
  - \( \text{mr} = X; \)
  - \( \text{IRld} = 0; \)
  - \( \text{ALUmaEN} = 0; \)
  - \( \text{mw} = 0; \)
  - \( \text{RegBmdEN} = 0; \)
  - \( \text{srcA} = \text{“A”} = 0; \)
  - \( \text{srcB} = \text{“B”} = 2'b00; \)
  - \( \text{op} = \text{“+”} = 6'b000001; \)
  - \( \text{regWrite} = 1; \)
  - \( \text{wrDataSel} = \text{“ALU”} = 0; \)
  - \( \text{wrRegSel} = \text{“rd”} = 1; \)

- **But, also . . .**
  - \( \text{PCld} = 0; \)
  - \( \text{PCsel} = X; \)
  - \( \text{MBRld} = X; \)
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, ALUmaEN)
    - 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 ← ALUout — 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 ← ALUout — 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, busses, 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
- **step 2**: decode
- **step 3**: execute

- IR ← mem[PC]
- PC ← PC + 1;
- A ← rs
- B ← rt
- 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
### 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;
```

### 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?

```
branch based on value of Inst[31:26] and Inst[5:0]
```
FSM controller for CPU (instruction execution)

- For add instruction
  - configure ALU and store result in register
    $$rd \leftarrow A + B$$
  - 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

- reset
  - Fetch
    - instruction fetch
  - Decode
    - instruction decode
  - add
    - instruction execution
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

Registers and 3-state drivers

```verilog
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
```
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'hzzzzzzzzz;
endmodule

Memory – initial contents (test fixture)

parameter ALU = 6'h00; // op = 0
parameter LW = 6'h23; // op = 35
parameter SW = 6'h2b; // op = 43
parameter BEQ = 6'h04; // op = 4
parameter ADDI = 6'h08; // op = 8
parameter J = 6'h02; // op = 2
parameter HALT = 6'h3f; // op = 63
parameter ADD = 6'h20; // funct = 32
parameter SUB = 6'h22; // funct = 34
parameter AND = 6'h24; // funct = 36
parameter OR = 6'h25; // funct = 37
parameter SLT = 6'h2a; // funct = 42

initial begin
  memory[8'h00] = {ADDI, rz, r1, 16'h0000}; // r1 = 0
  memory[8'h01] = {ADDI, rz, r2, 16'h0001}; // r2 = 1
  memory[8'h02] = {LW, rz, r0, 16'h00fe}; // r0 = mem [ rz + 254 ]
  memory[8'h03] = {ALU, rz, r1, r3, shftX, SLT}; // r3 = ( rz < r0 )
  memory[8'h04] = {BEQ, r3, rz, 16'h0009}; // if ( r3 == rz ) goto exit2
  memory[8'h05] = {BEQ, rz, rz, 16'h0002}; // if ( rz == rz ) goto entry /* goto entry
  memory[8'h06] = {ALU, r1, r2, r1, shftX, ADD}; // loop: r1 = r1 + r2
  memory[8'h07] = {ADDI, r0, r0, 16'hffff}; // r0 = r0 + ( -1 )
  memory[8'h08] = {BEQ, r0, rz, 16'h0000}; // entry: if ( r0 == rz ) goto entry1
  memory[8'h09] = {ALU, r2, r1, r2, shftX, ADD}; // r2 = r2 + r1
  memory[8'h0a] = {ADDI, r0, r0, 16'hffff}; // r0 = r0 + ( -1 )
  memory[8'h0b] = {BEQ, r0, rz, 16'h0000}; // if ( r0 == rz ) goto exit2
  memory[8'h0c] = {J, 26'h00000006}; // goto loop
  memory[8'h0d] = {ADD, r1, r2, r2, shftX, OR}; // exit1: r2 = r1 | r2 *= r2 = r1
  memory[8'h0e] = {SW, r1, r2, 16'h00ff}; // exit: mem [ r2 + 255 ] = r2
  memory[8'h0f] = {HALT, 26'hxxxxxxx}; // halt
end

parameter shftX = 5'hxx;
parameter r0 = 5'h00;
parameter r1 = 5'h01;
parameter r2 = 5'h02;
parameter r3 = 5'h03;
parameter rz = 5'h1f;

initial begin
  memory[8'h00] = {ADDI, rz, r1, 16'h0000}; // r1 = 0
  memory[8'h01] = {ADDI, rz, r2, 16'h0001}; // r2 = 1
  memory[8'h02] = {LW, rz, r0, 16'h00fe}; // r0 = mem [ rz + 254 ]
  memory[8'h03] = {ALU, rz, r0, r3, shftX, SLT}; // r3 = ( rz < r0 )
  memory[8'h04] = {BEQ, r3, rz, 16'h0009}; // if ( r3 == rz ) goto exit2
  memory[8'h05] = {BEQ, rz, rz, 16'h0002}; // if ( rz == rz ) goto entry /* goto entry
  memory[8'h06] = {ALU, r1, r2, r1, shftX, ADD}; // loop: r1 = r1 + r2
  memory[8'h07] = {ADDI, r0, r0, 16'hffff}; // r0 = r0 + ( -1 )
  memory[8'h08] = {BEQ, r0, rz, 16'h0000}; // entry: if ( r0 == rz ) goto entry1
  memory[8'h09] = {ALU, r2, r1, r2, shftX, ADD}; // r2 = r2 + r1
  memory[8'h0a] = {ADDI, r0, r0, 16'hffff}; // r0 = r0 + ( -1 )
  memory[8'h0b] = {BEQ, r0, rz, 16'h0000}; // if ( r0 == rz ) goto exit2
  memory[8'h0c] = {J, 26'h00000006}; // goto loop
  memory[8'h0d] = {ADD, r1, r2, r2, shftX, OR}; // exit1: r2 = r1 | r2 *= r2 = r1
  memory[8'h0e] = {SW, r1, r2, 16'h00ff}; // exit: mem [ r2 + 255 ] = r2
  memory[8'h0f] = {HALT, 26'hxxxxxxx}; // halt
end

// this is the input N
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 [5:0] op;
input srcA;
input [1:0] srcB;
output [31:0] ALUout;
output zero, neg;

wire [31:0] A;
reg [31:0] B;
reg result;
reg zero;
reg neg;

assign A = (srcA) ? PC : RegA;
always @(Inst or RegB or srcB) begin
  case (srcB)
    2'b00: B = RegB;
    2'b01: B = 32'h00000000;
    2'b10: B = {Inst[15], Inst[15], Inst[15], Inst[15], Inst[15], Inst[15],
                Inst[15], Inst[15], Inst[15], Inst[15], Inst[15], Inst[15],
                Inst[15], Inst[15], Inst[15], Inst[15], Inst[15:0]};
    2'b11: B = 32'h00000001;
  endcase
end

always @(A or B or op) begin
  case (op)
    6'b000001: result = A + B;
    6'b000010: result = A - B;
    6'b001000: result = A ; B;
    6'b010000: result = A ; B;
    default: result = 32'hxxxxxxxx;
  endcase
  zero = (result == 32'h00000000);
  neg = result[31];
end

assign ALUout = result;
endmodule

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
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
```

Estimating performance

- **Recall basic constraint equations:**
  - $T_{\text{period}} > T_{\text{FFprop}} + T_{\text{delay}} + T_{\text{setup}}$
  - $T_{\text{prop}} > T_{\text{hold}}$
- 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 $T_{\text{FFprop}}$ and $T_{\text{setup}}$ are the same for all FFs:
    $\max(T_{\text{period}}) = T_{\text{FFprop}} + \max(T_{\text{delay}}) + T_{\text{setup}}$
Estimating performance for “fetch” and “decode” cycles

- Max(T\(_{\text{delay}}\)) = Max of
  - \(T_{\text{3state}} + T_{\text{memoryread}}\)
  - \(T_{\text{Amux}} + T_{\text{ALU}} + T_{\text{PCmux}}\)
  - \(T_{\text{RegFileRead}}\)
  - \(T_{\text{controller}}\)

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

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

- 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
  - $T_{\text{wire}}$ 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