CSE471—Computer Design and Organization

Prof. Steven Burns *Autumn 1994*

Lecture Notes: Thirteenth Day

Variable Clock MIPS

- We can modify the single cycle simulator to run with a variable speed clock.

```
module clock( clk, op);
   input [5:0] op;
   output clk; reg clk;
   initial begin
      clk = 0; #(41)
      repeat (45) begin
         clk = 1; #10; clk = 0; #1;
         if      ( op == 6'b000000 )
            #20 ;
         else if ( op == 6'b100011 )
            #30 ;
         else if ( op == 6'b101011 )
            #25 ;
         else if ( op == 6'b000100 )
            #15 ;
         else if ( op == 6'b000010 )
            #0 ;
      end
      #(41)
      $stop;
   end
endmodule
```

Performance

- The time to run the following program decreases from 1599 units to 1214 units with a variable clock.

```
        .text
        lw      $2,four
        lw      $3,ubound
        lw      $6,one
        add     $4,$0,$0
        add     $5,$0,$0
top:    slt     $8,$4,$3
        beq     $8,$0,loop
        add     $7,$5,$6
        sw      $7, 0($4)
        add     $5,$0,$6
        add     $6,$0,$7
        add     $4,$4,$2
        beq     $0, $0, top
loop:   beq     $0, $0, loop
        .data
four:   .word 0x00000004
ubound: .word 0x00000010
one:    .word 0x00000001
```

Multiple Clock MIPS

- Another, more practical option is to break instructions into multiple clocks.
- This allows hardware reuse.
- It doesn't guarantee improved performance. In fact, on the previous program, 1694 units are required to complete the program.
- Why?

Multiple Clock MIPS Control

- You need to build a structural (implement the block diagram) simulator for the multiple clock MIPS. I'll show you one way to do the control.

```
always @(posedge clk) begin
   case (s)
```

```
        0: s = 1;
        1: if      ( Rtype)        s = 6;
           else if ( Load | Store) s = 2;
           else if ( Branch)       s = 8;
           else if ( Jump)         s = 9;
           else $display( "Illegal Opcode %b", Op);
        2: if ( Load)              s = 3;
           else                    s = 5;
        3: s = 4;
        4: s = 0;
        5: s = 0;
        6: s = 7;
        7: s = 0;
        8: s = 0;
        9: s = 0;
        endcase
     end
```

Moore Machine Outputs

```
always @(s)
begin
  MemRead=0; IRWrite=0; PCWrite=0; TargetWrite=0;
  MemWrite=0; RegWrite=0; PCWriteCond=0;
  ALUSelA=1'bx; IorD=1'bx; RegDst=1'bx; MemtoReg=1'bx;
  PCSource=2'bxx; ALUOp=2'bxx; ALUSelB=2'bxx;
  case (s)
  0: begin MemRead=1; IRWrite=1; ALUSelB=2'b01; ALUOp=2'b00;
           PCWrite=1; ALUSelA=0; IorD=0; PCSource=2'b00; end
  1: begin ALUSelB=2'b11; ALUOp=2'b00; TargetWrite=1;
           ALUSelA=0; end
  2: begin ALUSelA=1; ALUSelB=2'b10; ALUOp=2'b00; end
  3: begin MemRead=1; ALUSelA=1; IorD=1; ALUSelB=2'b10;
           ALUOp=2'b00; end
  4: begin MemRead=1; ALUSelA=1; IorD=1; ALUSelB=2'b10;
           ALUOp=2'b00; RegWrite=1; MemtoReg=1; RegDst=0; end
  5: begin ALUSelA=1; ALUSelB=2'b10; ALUOp=2'b00; MemWrite=1; end
  6: begin ALUSelA=1; ALUSelB=2'b00; ALUOp=2'b10; end
  7: begin ALUSelA=1; ALUSelB=2'b00; ALUOp=2'b10; RegWrite=1;
```

```
          RegDst=1; MemtoReg=0; end
  8: begin ALUSelA=1; ALUSelB=2'b00; ALUOp=2'b01; PCSource=2'b01;
          PCWriteCond=1; end
  9: begin PCWrite=1; PCSource=2'b10; end
  endcase
end
```

Microprogramming

- Still the best way to specify the control of complex tasks
- Applications
  - CISC processors; VAX, IBM 360, iax86
  - Cheap implementations of RISC ISAs
  - Control for glue chips, e.g., the MAGIC chip in Stanford's FLASH multicomputer.

What is Microprogramming?

- A structured way to produce the control FSM for a processor
- Similar to assembler programming although it is even more detailed.
- Designer specifies sequencing between micro-states and the value of control signals for the datapath at each micro-state. This information is encoded in *micro-instructions*.
- This microprogramming is performed at an abstract level. A microassembler translates this representation into a representation suitable for hardware implementation of the control.

Micro-Instruction Sequencing

- The most common next state is the one plus the present micro program counter (μPC). We need an incrementer to do this.
- All other next sequencing control is done through dispatch tables.
- Sequencing control can cause a jump to any micro-state based on the current opcode. This can be implemented with a ROM or a PLA. For the MIPS, we need a 2^6 or 64 micro-address-word ROM for each dispatch table. PLAs can be much smaller but are less flexible.

Control Signals

- The control signals for the datapath are set depending on the current micro-state.
- For convenience, mnemonics are used. Sometimes, a particular mnemonic specifies that several signals (not just one) are set to a particular pattern.
- This is called *encoding*. Sometimes the representation of this translation from the encoded form to the real signals is called nano-code (a set of nano-instructions).

Vertical vs. Horizontal Microcode

- This distinction is meant to represent the layout of the control store.
- Microcode is horizontal if there are relatively more control signals and fewer microinstructions.
- Microcode is vertical if there are relatively fewer control signals and more microinstructions.
- Vertical microcode usually leads to a larger CPI but less hardware.

Maximally- vs. Minimally-encoded

- The Horizontal vs. Vertical distinction is very subjective.
- The marketing people for the 8086 called their 21-bit wide microcode horizontal because it was more horizontal than similar machines of the era.
- A better, though slightly different distinction is the degree of encoding of the control signals.
- *Maximally-encoded* microcode loosely corresponds to vertical microcode. *Minimally-encoded* microcode loosely corresponds to horizontal microcode.

Microprogramming for the MIPS Subset

- The microinstruction format consists of seven, possibly encoded, control signal fields and one sequencing field.
- The fields are named: ALU control, SRC1, SRC2, ALU destination, Memory, Memory register, PCWrite control, and Sequencing.
- Example values for the PCwrite control field are: `ALU`, `Target-cond`, and `jump address`.
- These mnemonics correspond to setting the control signals *PCSource*, *PCWriteCond*, and *PCWrite* to appropriate patterns. These patterns are: `00 0 1`, `01 1 0`, and `10 0 1`, respectively.

Let's Add a Complex Instruction

- Consider implementing an instruction especially for indexing through a for-loop. We'll call it `bxne`. The intended operation is: increment and replace the value of a register rs, compare the new value of rs to the value of register rt, and then perform a relative branch if not equal.
- The instruction is in I-format, just like `bne`.
- What new datapath facilities to we need to implement this new instruction?
- What microinstruction sequence should we produce for our new instruction?