Mini-MIPS

From Weste/Harris
CMOS VLSI Design

1

Based on MIPS

- In fact, it's based on the multi-cycle MIPS from Patterson and Hennessy
  - Your CS/EE 3810 book...
- 8-bit version
  - 8-bit data and address
  - 32-bit instruction format
  - 8 registers numbered $0-$7
    - $0 is hardwired to the value 0

2

Instruction Set

Table 1.7 MIPS instruction set (subset supported)

Instruction	Function		Encoding	op	funct
`add $1, $2, $3`	addition:	`$1 <- $2 + $3`	R	000000	100000
`sub $1, $2, $3`	subtraction:	`$1 <- $2 – $3`	R	000000	100010
`and $1, $2, $3`	bitwise and:	`$1 <- $2 and $3`	R	000000	100100
`or $1, $2, $3`	bitwise or:	`$1 <- $2 or $3`	R	000000	100101
`slt $1, $2, $3`	set less than:	`$1 <- 1 if $2 < $3` `$1 <- 0 otherwise`	R	000000	101010
`addi $1, $2, imm`	add immediate:	`$1 <- $2 + imm`	I	001000	n/a
`beq $1, $2, imm`	branch if equal:	`PC <- PC + imm`[a]	I	000100	n/a
`j destination`	jump:	`PC <- destination`[a]	J	000010	n/a
`lb $1, imm($2)`	load byte:	`$1 <- mem[$2 + imm]`	I	100000	n/a
`sb $1, imm($2)`	store byte:	`mem[$2 + imm] <- $1`	I	101000	n/a

a. Technically, MIPS addresses specify bytes. Instructions require a four-byte word and must begin at addresses that are a multiple of four. To most effectively use instruction bits in the full 32-bit MIPS architecture, branch and jump constants are specified in words and must be multiplied by four (shifted left two bits) to be converted to byte addresses.

3

Instruction Encoding

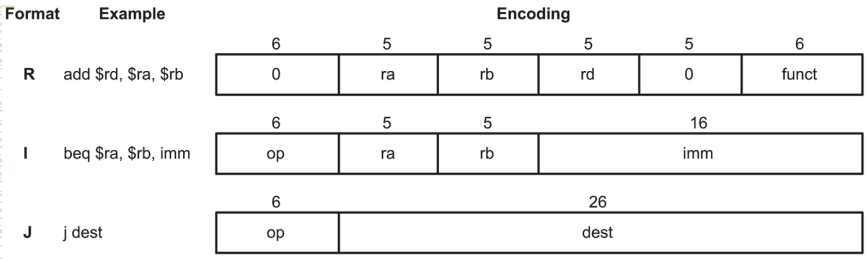


FIG 1.49 Instruction encoding formats

4

Fibonacci C-Code

```
int fib(void)
{
   int n = 8;             /* compute nth Fibonacci number */
   int f1 = 1, f2 = -1;   /* last two Fibonacci numbers */

   while (n != 0) {       /* count down to n = 0 */
     f1 = f1 + f2;
     f2 = f1 - f2;
     n = n - 1;
   }
   return f1;
}
```

FIG 1.50 C code for Fibonacci program

5

Fibonacci C-Code

```
int fib(void)
{
   int n = 8;             /* compute nth Fibonacci number */
   int f1 = 1, f2 = -1;   /* last two Fibonacci numbers */

   while (n != 0) {       /* count down to n = 0 */
     f1 = f1 + f2;
     f2 = f1 - f2;
     n = n - 1;
   }
   return f1;
}
```

FIG 1.50 C code for Fibonacci program

```
Cycle 1: f1 = 1 + (-1) = 0, f2 = 0 - (-1) = 1
Cycle 2: f1 = 0 + 1 = 1, f2 = 1 - 1 = 0
Cycle 3: f1 = 1 + 0 = 1, f2 = 1 - 0 = 1
Cycle 4: f1 = 1 + 1 = 2, f2 = 2 - 1 = 1
Cycle 5: f1 = 2 + 1 = 3, f2 = 3 - 1 = 2
Cycle 6: f1 = 3 + 2 = 5, f2 = 5 - 2 = 3
```

6

Fibonacci Assembly Code

```
# fib.asm
# Register usage: $3: n $4: f1 $5: f2
# return value written to address 255
fib:  addi $3, $0, 8       # initialize n=8
      addi $4, $0, 1       # initialize f1 = 1
      addi $5, $0, -1      # initialize f2 = -1
loop: beq $3, $0, end      # Done with loop if n = 0
      add $4, $4, $5       # f1 = f1 + f2
      sub $5, $4, $5       # f2 = f1 - f2
      addi $3, $3, -1      # n = n - 1
      j loop               # repeat until done
end:  sb $4, 255($0)       # store result in address 255
```

FIG 1.51 Assembly language code for Fibonacci program

Compute 8th Fibonacci number (8'd13 or 8'h0D)
Store that number in memory location 255

7

Fibonacci Machine Code

Instruction	Binary Encoding	Hexadecimal Encoding
addi $3, $0, 8	001000 00000 00011 0000000000001000	20030008
addi $4, $0, 1	001000 00000 00100 0000000000000001	20040001
addi $5, $0, -1	001000 00000 00101 1111111111111111	2005ffff
beq $3, $0, end	000100 00011 00000 0000000000000101	1060000 4
add $4, $4, $5	000000 00100 00101 00100 00000 100000	00852020
sub $5, $4, $5	000000 00100 00101 00101 00000 100010	00852822
addi $3, $3, -1	001000 00011 00011 1111111111111111	2063ffff
j loop	000010 00000000000000000000000011	08000003
sb $4, 255($0)	101000 00000 00100 0000000011111111	a00400ff

FIG 1.52 Machine language code for Fibonacci program

Assembly Code

Machine Code

8

Architecture

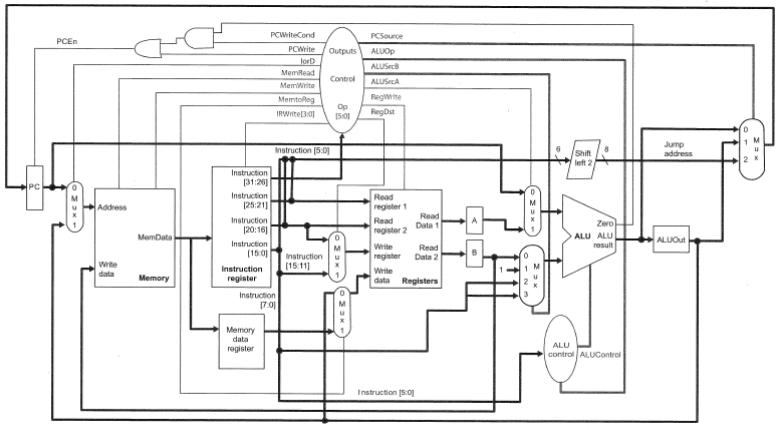


FIG 1.53 Multicycle MIPS microarchitecture. Reprinted from [Patterson04] with permission from Elsevier.

9

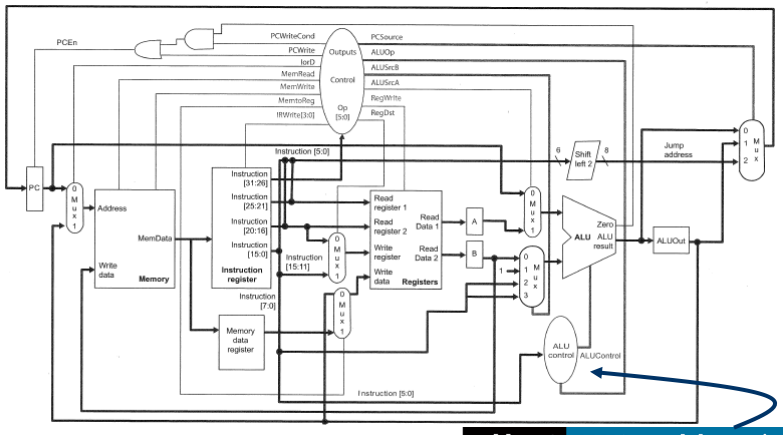


FIG 1.53 Multicycle MIPS microarchitecture. Reprinted from [Patterson04]

address
beq
R-type

Won't happen…

Table 1.8 ALUControl determination

aluop	funct	alucontrol	Meaning
00	x	010	ADD
01	x	110	SUB
10	100000	010	ADD
10	100010	110	SUB
10	100100	000	AND
10	100101	001	OR
10	101010	111	SLT
11	x	x	undefined

10

Another View

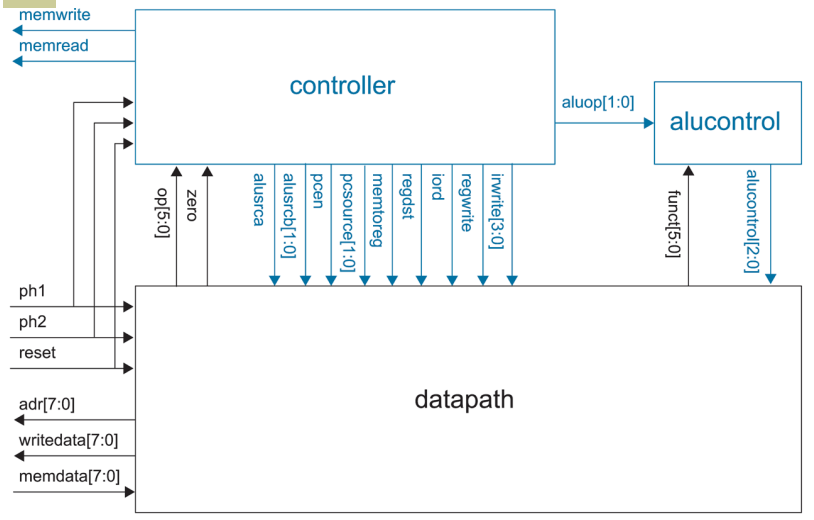
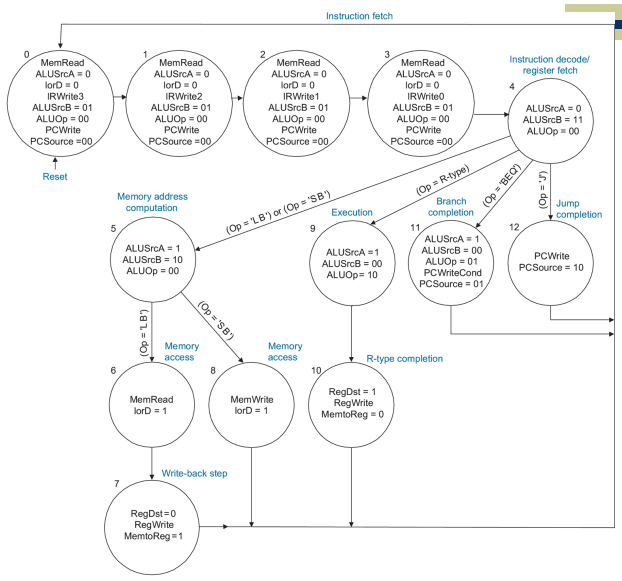


FIG 1.56 Top-level MIPS block diagram

11

Control FSM



CS/EE 3710

FIG 1.54 Multicycle MIPS control FSM. Reprinted from [Patterson04] with permission from Elsevier.

12

Connection to External Memory

Table 1.9 Top-level inputs and outputs

Inputs	Outputs
ph1	adr[7:0]
ph2	writedata[7:0]
reset	memread
memdata[7:0]	memwrite

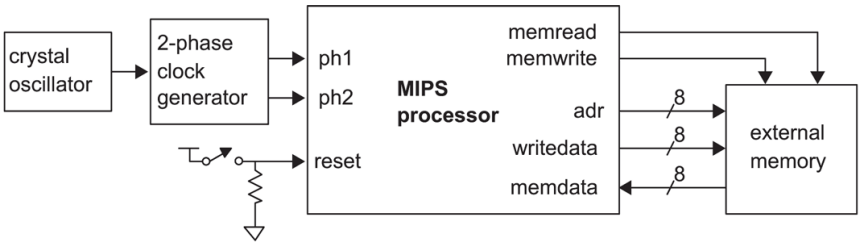


FIG 1.55 MIPS computer system

13

External Memory from Book

```
// external memory accessed by MIPS
module exmemory #(parameter WIDTH = 8)
  (input                clk,
  input                 memwrite,
  input      [WIDTH-1:0] adr, writedata,
  output reg [WIDTH-1:0] memdata);

  reg  [31:0] RAM [(1<<WIDTH-2)-1:0];
  wire [31:0] word;

  // Initialize memory with program
     initial $readmemh("memfile.dat",RAM);

  // read and write bytes from 32-bit word
  always @(posedge clk)
    if(memwrite)
      case (adr[1:0])
        2'b00: RAM[adr>>2][7:0] <= writedata;
        2'b01: RAM[adr>>2][15:8] <= writedata;
        2'b10: RAM[adr>>2][23:16] <= writedata;
        2'b11: RAM[adr>>2][31:24] <= writedata;
      endcase

  assign word = RAM[adr>>2];
    always @(*)
      case (adr[1:0])
        2'b00: memdata <= word[7:0];
        2'b01: memdata <= word[15:8];
        2'b10: memdata <= word[23:16];
        2'b11: memdata <= word[31:24];
      endcase
  endmodule
```

Notes:

- Endianess is fixed here
- Writes are on posedge clk
- Reads are asynchronous
- This is a 32-bit wide RAM
- With 64 locations
- But with an 8-bit interface...

14

External memory on FPGA

```
module exmem #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=8)
      ( input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] addr,
        input                    we, clk,
        output [(DATA_WIDTH-1):0] q );
    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
    // Variable to hold the read address
    reg [ADDR_WIDTH-1:0] addr_reg;
    always @ (posedge clk)
      begin
            // Write
            if (we) ram[addr] <= data;
            // register to hold the read address
            addr_reg <= addr;
      end
     assign q = ram[addr_reg];
endmodule // exmem
```

- This is synthesized to a Block RAM on the FPGA
- It's 8-bits wide
- With 256 locations
- Both writes and reads are clocked

15

exmem.v

```
module exmem #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=8)
      ( input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] addr,
        input                    we, clk,
        output [(DATA_WIDTH-1):0] q );
    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
    // Variable to hold the read address
    reg [ADDR_WIDTH-1:0] addr_reg;
    always @ (posedge clk)
      begin
            // Write
            if (we) ram[addr] <= data;
            // register to hold the read address
            addr_reg <= addr;
      end
     assign q = ram[addr_reg];
endmodule // exmem
```

This is synthesized to a Block RAM on the FPGA

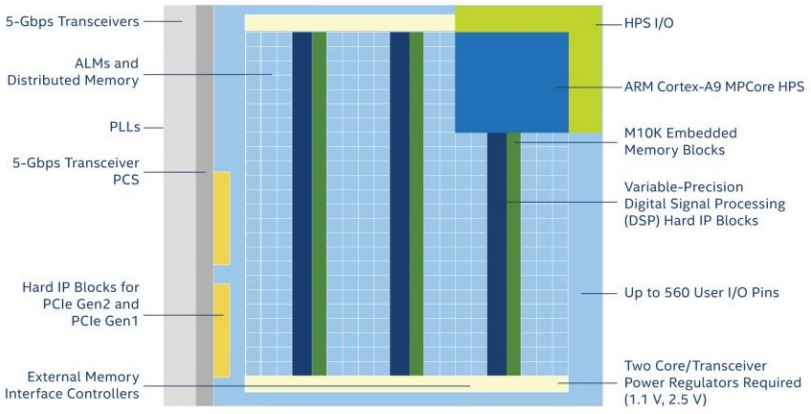
Note clock!

16

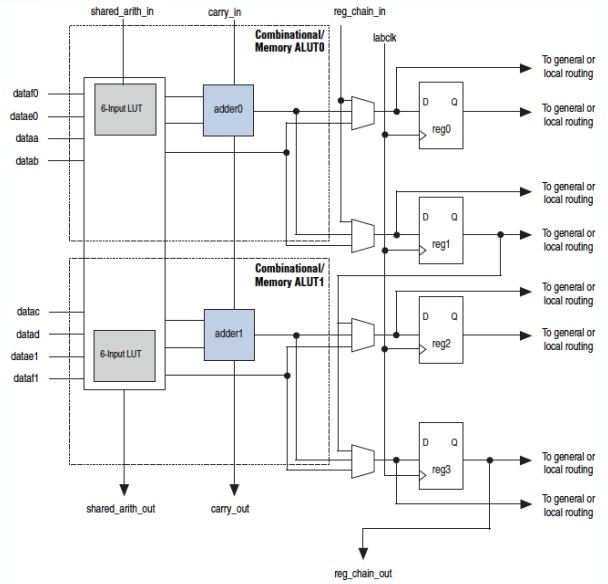
Memory Types on the FPGA

- DFFs available in Logic Array Blocks
  - Used mostly for local, unstructured memories
  - Used as DFFs in FSMs, local registers (regfile, flags, FSM control state variables, etc.)
- For more structured Memories, we use BLOCK RAMs (M10K)
  - Configurable w.r.t. operand word-lengths, and memory words
  - Use as RAMs, as well as ROMs

FPGA Overview



Adaptive Logic Modules (ALMs)



19

Cyclone V SE 5CSEMA5F31C6

Table 4. Maximum Resource Counts for Cyclone V E Devices

Resource		Member Code				
		A2	A4	A5	A7	A9
Logic Elements (LE) (K)		25	40	77	150	301
ALM		9,430	18,480	29,080	56,480	113,560
Register		37,736	73,840	116,320	225,920	454,240
Memory (Kb)	M10K	1,760	3,080	4,460	6,860	12,200
	MLAB	196	304	424	836	1,717
Variable-precision DSP Block		25	66	150	156	342
18 x 18 Multiplier		50	132	300	312	684
PLL		4	4	6	7	8
GPIO		224	224	240	480	480
LVDS	Transmitter	56	56	60	120	120
	Receiver	56	56	60	120	120
Hard Memory Controller		1	1	2	2	2

20

M10K RAM Blocks

- M10K = 10K bits = 10x1024 = 10,240 bits
  - 397 blocks = 397 x 10K bits total

Table 19. Supported Embedded Memory Block Configurations for Cyclone V Devices

This table lists the maximum configurations supported for the embedded memory blocks. The information is applicable only to the single-port RAM and ROM modes.

Memory Block	Depth (bits)	Programmable Width
MLAB	32	x16, x18, or x20
M10K	256	x40 or x32
	512	x20 or x16
	1K	x10 or x8
	2K	x5 or x4
	4K	x2
	8K	x1

Looking Ahead

- Project has 16bit words
  - 512x16 is one M10K configuration
  - 397 blocks = 203,264 words
  - 2^{16} = 65,535 words
  - 2^{18} = 262,144
- So, maybe you could use an 18-bit address with the 16-bit data word?
  - That's what the CR-16 does…

Synchronous RAM

Table 5. Write and Read Operations Triggering for Embedded Memory Blocks

This table lists the write and read operations triggering for various embedded memory blocks.

Embedded Memory Blocks	Write Operation	Read Operation
M10K	Rising clock edges	Rising clock edges

- Single Port: one read or one write per cycle
  - Synchronous with rising clock edge
- Dual Port
  - Two reads or two writes per cycle
  - Can even support different clocks on each port

exmem.v

```
module exmem #(parameter DATA_WIDTH=8, parameter ADDR_WIDTH=8)
      ( input [(DATA_WIDTH-1):0] data,
        input [(ADDR_WIDTH-1):0] addr,
        input                    we, clk,
        output [(DATA_WIDTH-1):0] q );
    // Declare the RAM variable
    reg [DATA_WIDTH-1:0] ram[2**ADDR_WIDTH-1:0];
    // Variable to hold the read address
    reg [ADDR_WIDTH-1:0] addr_reg;
    always @ (posedge clk)
      begin
            // Write
            if (we) ram[addr] <= data;
            // register to hold the read address
            addr_reg <= addr;
      end
     assign q = ram[addr_reg];
endmodule // exmem
```

Addr_reg is stored on rising edge

So, Read happens using stored address

Recall – Overall System

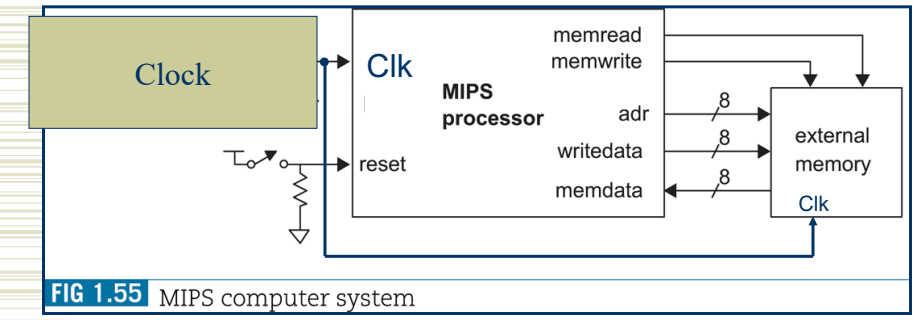


FIG 1.55 MIPS computer system

25

Recall – Overall System

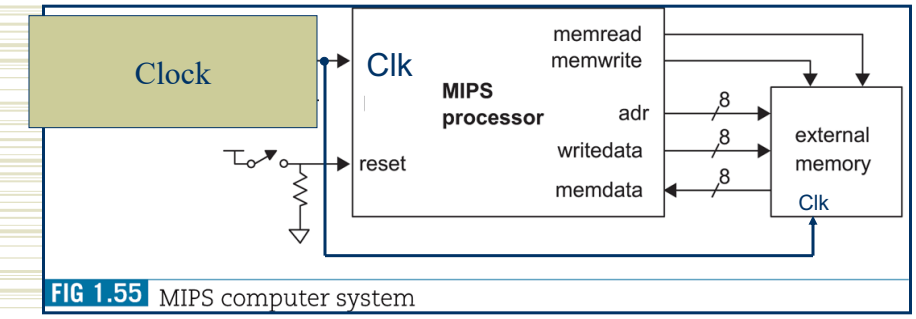


FIG 1.55 MIPS computer system

So, what are the implications of using a RAM that has both clocked reads and writes instead of clocked writes and async reads? (we'll come back to this question...)

26

mips Block Diagram

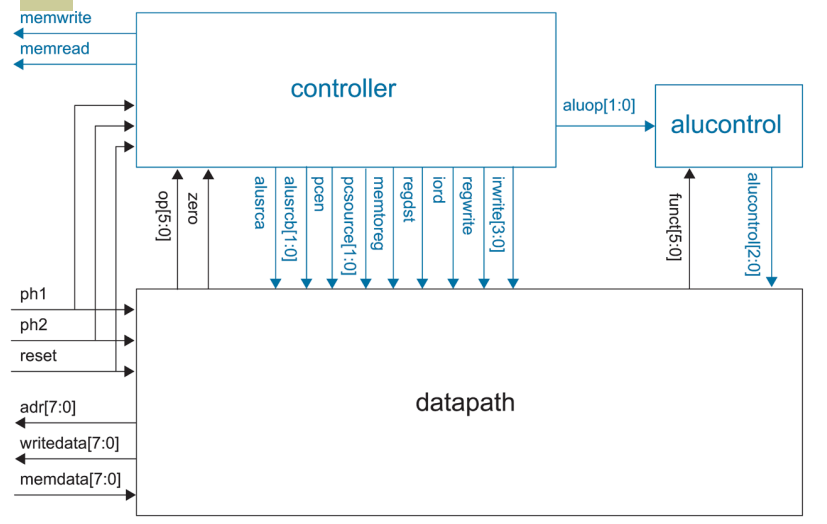


FIG 1.56 Top-level MIPS block diagram

27

mips.v

```
// simplified MIPS processor
module mips #(parameter WIDTH = 8, REGBITS = 3)
             (input            clk, reset,
              input  [WIDTH-1:0] memdata,
              output           memread, memwrite,
              output [WIDTH-1:0] adr, writedata);

  wire [31:0] instr;
  wire        zero, alusrca, memtoreg, iord, pcen, regwrite, regdst;
  wire [1:0]  aluop,pcsource,alusrcb;
  wire [3:0]  irwrite;
  wire [2:0]  alucont;

  controller  cont(clk, reset, instr[31:26], zero, memread, memwrite,
                   alusrca, memtoreg, iord, pcen, regwrite, regdst,
                   pcsource, alusrcb, aluop, irwrite);
  alucontrol  ac(aluop, instr[5:0], alucont);
  datapath    #(WIDTH, REGBITS)
              dp(clk, reset, memdata, alusrca, memtoreg, iord, pcen,
                 regwrite, regdst, pcsource, alusrcb, irwrite, alucont,
                 zero, instr, adr, writedata);
endmodule
```

28

Controller

```
module controller(input clk, reset,
                  input      [5:0] op,
                  input            zero,
                  output reg       memread, memwrite, alusrca, memtoreg, iord,
                  output           pcen,
                  output reg       regwrite, regdst,
                  output reg [1:0] pcsource, alusrcb, aluop,
                  output reg [3:0] irwrite);

   parameter   FETCH1  =  4'b0001;
   parameter   FETCH2  =  4'b0010;
   parameter   FETCH3  =  4'b0011;
   parameter   FETCH4  =  4'b0100;
   parameter   DECODE  =  4'b0101;
   parameter   MEMADR  =  4'b0110;
   parameter   LBRD    =  4'b0111;
   parameter   LBWR    =  4'b1000;
   parameter   SBWR    =  4'b1001;
   parameter   RTYPEEX =  4'b1010;
   parameter   RTYPEWR =  4'b1011;
   parameter   BEQEX   =  4'b1100;
   parameter   JEX     =  4'b1101;

   parameter   LB      =  6'b100000;
   parameter   SB      =  6'b101000;
   parameter   RTYPE   =  6'b0;
   parameter   BEQ     =  6'b000100;
   parameter   J       =  6'b000010;

   reg [3:0] state, nextstate;
   reg       pcwrite, pcwritecond;

   // state register
   always @(posedge clk)
      if(reset) state <= FETCH1;
      else state <= nextstate;
```

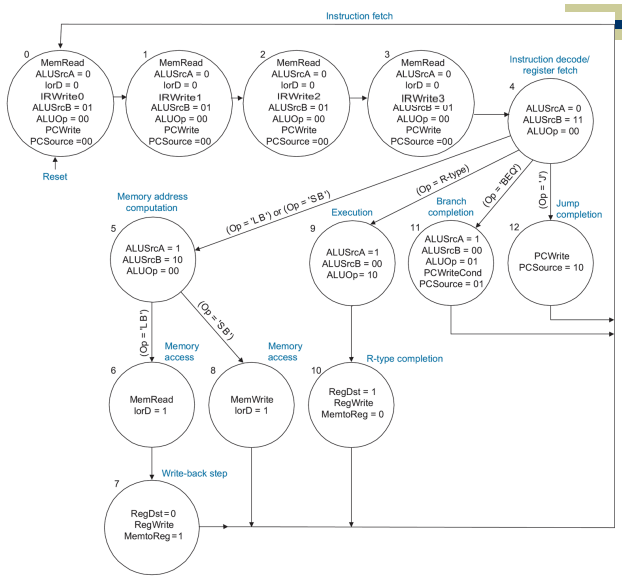
State Codes

Useful constants to compare against

State Register

29

Control FSM



CS/EE 3710

FIG 1.54 Multicycle MIPS control FSM. Reprinted from [Patterson04] with permission from Elsevier.

30

Next State Logic

```
// next state logic
always @(*)
   begin
      case(state)
         FETCH1:  nextstate <= FETCH2;
         FETCH2:  nextstate <= FETCH3;
         FETCH3:  nextstate <= FETCH4;
         FETCH4:  nextstate <= DECODE;
         DECODE:  case(op)
                     LB:      nextstate <= MEMADR;
                     SB:      nextstate <= MEMADR;
                     RTYPE:   nextstate <= RTYPEEX;
                     BEQ:     nextstate <= BEQEX;
                     J:       nextstate <= JEX;
                     default: nextstate <= FETCH1; // should never happen
                  endcase
         MEMADR:  case(op)
                     LB:      nextstate <= LBRD;
                     SB:      nextstate <= SBWR;
                     default: nextstate <= FETCH1; // should never happen
                  endcase
         LBRD:    nextstate <= LBWR;
         LBWR:    nextstate <= FETCH1;
         SBWR:    nextstate <= FETCH1;
         RTYPEEX: nextstate <= RTYPEWR;
         RTYPEWR: nextstate <= FETCH1;
         BEQEX:   nextstate <= FETCH1;
         JEX:     nextstate <= FETCH1;
         default: nextstate <= FETCH1; // should never happen
      endcase
   end
```


31

Output Logic

```
always @(*)
   begin
      // set all outputs to zero, then conditionally assert
      // just the appropriate ones
      irwrite <= 4'b0000;
      pcwrite <= 0; pcwritecond <= 0;
      regwrite <= 0; regdst <= 0;
      memread <= 0; memwrite <= 0;
      alusrca <= 0; alusrcb <= 2'b00; aluop <= 2'b00;
      pcsource <= 2'b00;
      iord <= 0; memtoreg <= 0;
      case(state)
         FETCH1:
            begin
               memread <= 1;
               irwrite <= 4'b0001; // changed to reflect new memory and
               alusrcb <= 2'b01;   // get the IR bits in the right spots
               pcwrite <= 1;       // FETCH 2,3,4 also changed...
            end
         FETCH2:
            begin
               memread <= 1;
               irwrite <= 4'b0010;
               alusrcb <= 2'b01;
               pcwrite <= 1;
            end
         FETCH3:
            begin
               memread <= 1;
               irwrite <= 4'b0100;
               alusrcb <= 2'b01;
               pcwrite <= 1;
            end
```

Very common way to deal with default values in combinational Always blocks

Continued for the other states...

32

Output Logic

```
            SBWR:
               begin
                  memwrite <= 1;
                  iord     <= 1;
               end
            RTYPEEX:
               begin
                  alusrca <= 1;
                  aluop   <= 2'b10;
               end
            RTYPEWR:
               begin
                  regdst   <= 1;
                  regwrite <= 1;
               end
            BEQEX:
               begin
                  alusrca     <= 1;
                  aluop       <= 2'b01;
                  pcwritecond <= 1;
                  pcsource    <= 2'b01;
               end
            JEX:
               begin
                  pcwrite  <= 1;
                  pcsource <= 2'b10;
               end
         endcase
      end
   assign pcen = pcwrite | (pcwritecond & zero); // program counter enable
endmodule
```

Two places to update the PC
pcwrite on jump
pcwritecond on BEQ

Why AND these two?

33

ALU Control

```
module alucontrol(input      [1:0] aluop,
                  input      [5:0] funct,
                  output reg [2:0] alucont);

   always @(*)
      case(aluop)
         2'b00: alucont <= 3'b010;  // add for lb/sb/addi
         2'b01: alucont <= 3'b110;  // sub (for beq)
         default: case(funct)       // R-Type instructions
                     6'b100000: alucont <= 3'b010; // add (for add)
                     6'b100010: alucont <= 3'b110; // subtract (for sub)
                     6'b100100: alucont <= 3'b000; // logical and (for and)
                     6'b100101: alucont <= 3'b001; // logical or (for or)
                     6'b101010: alucont <= 3'b111; // set on less (for slt)
                     default:   alucont <= 3'b101; // should never happen
                  endcase
      endcase
endmodule
```

Table 1.8 ALUControl determination

	aluop	funct	alucontrol	Meaning
address	00	x	010	ADD
beq	01	x	110	SUB
R-type	10	100000	010	ADD
	10	100010	110	SUB
	10	100100	000	AND
	10	100101	001	OR
	10	101010	111	SLT
Won't happen…	11	x	x	undefined

34

ALU

```
module alu #(parameter WIDTH = 8)
            (input      [WIDTH-1:0] a, b,
             input      [2:0]       alucont,
             output reg [WIDTH-1:0] result);

   wire     [WIDTH-1:0] b2, sum, slt;

   assign b2 = alucont[2] ? ~b:b;
   assign sum = a + b2 + alucont[2];
   // slt should be 1 if most significant bit of sum is 1
   assign slt = sum[WIDTH-1];

   always@(*)
      case(alucont[1:0])
         2'b00: result <= a & b;
         2'b01: result <= a | b;
         2'b10: result <= sum;
         2'b11: result <= slt;
      endcase
endmodule
```

Invert b if subtract...

add is a + b
sub is a + ~b +1

subtract on slt
then check if answer is negative

35

zerodetect

```
module zerodetect #(parameter WIDTH = 8)
                   (input [WIDTH-1:0] a,
                    output            y);

   assign y = (a==0);
endmodule
```


36

Register File

```
module regfile #(parameter WIDTH = 8, REGBITS = 3)
                (input                 clk,
                 input                 regwrite,
                 input  [REGBITS-1:0] ra1, ra2, wa,
                 input  [WIDTH-1:0]    wd,
                 output [WIDTH-1:0]    rd1, rd2);

   reg  [WIDTH-1:0] RAM [(1<<REGBITS)-1:0];

   // three ported register file
   // read two ports combinationally
   // write third port on rising edge of clock
   // register 0 hardwired to 0
   always @(posedge clk)
      if (regwrite) RAM[wa] <= wd;

   assign rd1 = ra1 ? RAM[ra1] : 0;
   assign rd2 = ra2 ? RAM[ra2] : 0;
endmodule
```

What is this synthesized into?

Datapath

```
module datapath #(parameter WIDTH = 8, REGBITS = 3)
                 (input                 clk, reset,
                  input  [WIDTH-1:0] memdata,
                  input                 alusrca, memtoreg, iord,
                  input                 pcen, regwrite, regdst,
                  input  [1:0]          pcsource, alusrcb,
                  input  [3:0]          irwrite,
                  input  [2:0]          alucont,
                  output                zero,
                  output [31:0]         instr,
                  output [WIDTH-1:0] adr, writedata);

   // the size of the parameters must be changed to match the WIDTH parameter
   localparam CONST_ZERO = 8'b0;
   localparam CONST_ONE =  8'b1;

   wire [REGBITS-1:0] ra1, ra2, wa;
   wire [WIDTH-1:0]   pc, nextpc, md, rd1, rd2, wd, a, src1, src2, aluresult,
                      aluout, constx4;

   // shift left constant field by 2
   assign constx4 = {instr[WIDTH-3:0],2'b00};

   // register file address fields
   assign ra1 = instr[REGBITS+20:21];
   assign ra2 = instr[REGBITS+15:16];
   mux2       #(REGBITS) regmux(instr[REGBITS+15:16], instr[REGBITS+10:11], regdst, wa);

   // independent of bit width, load instruction into four
   // 8-bit registers over four cycles
   flopen     #(8)      ir0(clk, irwrite[0], memdata[7:0], instr[7:0]);
   flopen     #(8)      ir1(clk, irwrite[1], memdata[7:0], instr[15:8]);
   flopen     #(8)      ir2(clk, irwrite[2], memdata[7:0], instr[23:16]);
   flopen     #(8)      ir3(clk, irwrite[3], memdata[7:0], instr[31:24]);
```

Fairly complex...

Not really, but it does have lots of registers instantiated directly

It also instantiates muxes...

Instruction Register

Datapath continued

```
// datapath
flopenr    #(WIDTH)  pcreg(clk, reset, pcen, nextpc, pc);
flop       #(WIDTH)  mdr(clk, memdata, md);
flop       #(WIDTH)  areg(clk, rd1, a);
flop       #(WIDTH)  wrd(clk, rd2, writedata);
flop       #(WIDTH)  res(clk, aluresult, aluout);
mux2       #(WIDTH)  adrmux(pc, aluout, iord, adr);
mux2       #(WIDTH)  src1mux(pc, a, alusrca, src1);
mux4       #(WIDTH)  src2mux(writedata, CONST_ONE, instr[WIDTH-1:0],
                             constx4, alusrcb, src2);
mux4       #(WIDTH)  pcmux(aluresult, aluout, constx4, CONST_ZERO, pcsource, nextpc);
mux2       #(WIDTH)  wdmux(aluout, md, memtoreg, wd);
regfile    #(WIDTH,REGBITS) rf(clk, regwrite, ra1, ra2, wa, wd, rd1, rd2);
alu        #(WIDTH) alunit(src1, src2, alucont, aluresult);
zerodetect #(WIDTH) zd(aluresult, zero);
endmodule
```

RF and ALU

Flops and muxes...

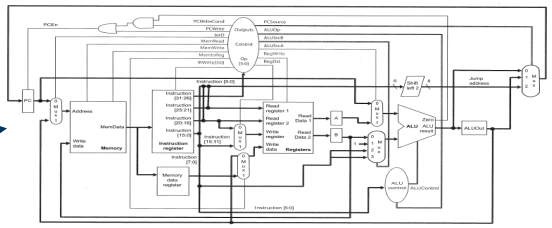


FIG 1.5.5 Multicycle MIPS microarchitecture. Reprinted from [Patterson04] with permission from Elsevier.

Flops and MUXes

```
module flop #(parameter WIDTH = 8)
             (input                  clk,
              input      [WIDTH-1:0] d,
              output reg [WIDTH-1:0] q);

   always @(posedge clk)
      q <= d;
endmodule

module flopen #(parameter WIDTH = 8)
               (input                  clk, en,
                input      [WIDTH-1:0] d,
                output reg [WIDTH-1:0] q);

   always @(posedge clk)
      if (en) q <= d;
endmodule

module flopenr #(parameter WIDTH = 8)
               (input                  clk, reset, en,
                input      [WIDTH-1:0] d,
                output reg [WIDTH-1:0] q);

   always @(posedge clk)
      if      (reset) q <= 0;
      else if (en)    q <= d;
endmodule
```

```
module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

   assign y = s ? d1 : d0;
endmodule

module mux4 #(parameter WIDTH = 8)
             (input      [WIDTH-1:0] d0, d1, d2, d3,
              input      [1:0]       s,
              output reg [WIDTH-1:0] y);

   always @(*)
      case(s)
         2'b00: y <= d0;
         2'b01: y <= d1;
         2'b10: y <= d2;
         2'b11: y <= d3;
      endcase
endmodule
```

Back to the Memory Question

- What are the implications of using RAM that is clocked on both write and read?
  - Book version was async read
  - So, let's look at the sequence of events that happen to read the instruction
  - Four steps – read four bytes and put them in four slots in the 32-bit instruction register (IR)

Instruction Fetch

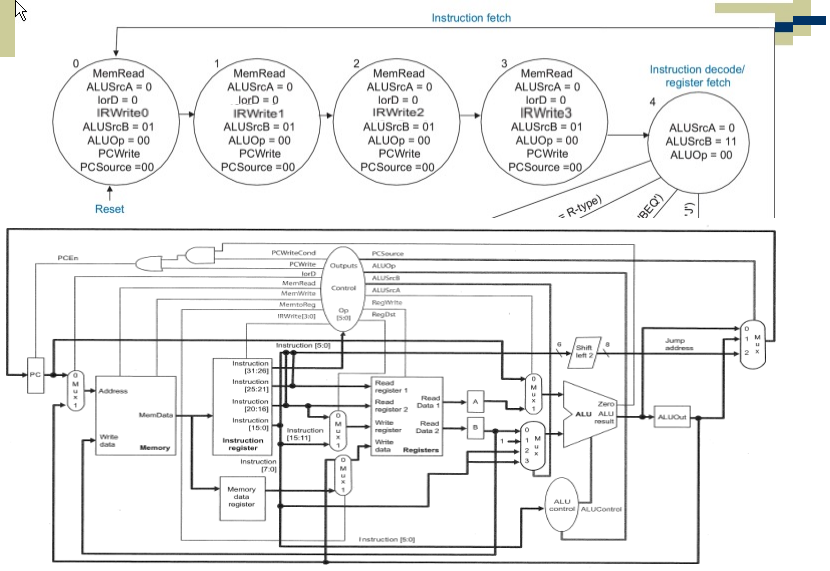


FIG 1.53 Multicycle MIPS microarchitecture. Reprinted from [Patterson04] with permission from Elsevier.

Instruction Fetch

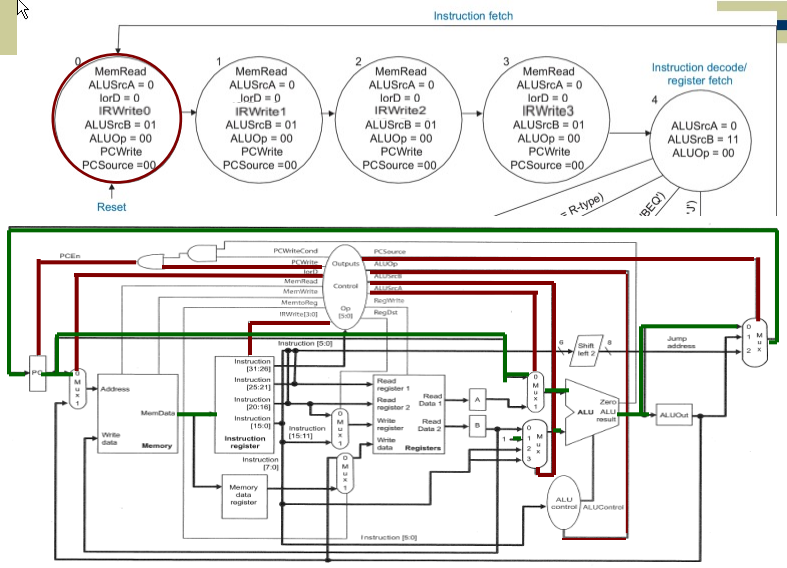


FIG 1.53 Multicycle MIPS microarchitecture. Reprinted from [Patterson04] with permission from Elsevier.

43

Instruction Fetch

```
// independent of bit width, load instruction into four
// 8-bit registers over four cycles
flopen    #(8)    ir0(clk, irwrite[0], memdata[7:0], instr[7:0]);
flopen    #(8)    ir1(clk, irwrite[1], memdata[7:0], instr[15:8]);
flopen    #(8)    ir2(clk, irwrite[2], memdata[7:0], instr[23:16]);
flopen    #(8)    ir3(clk, irwrite[3], memdata[7:0], instr[31:24]);
```

Instruction Register

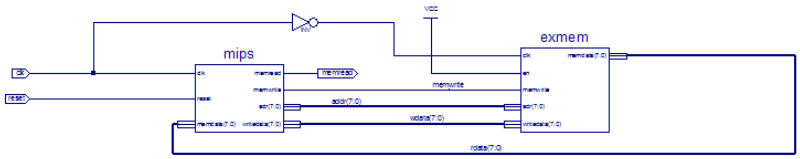
```
module flopen #(parameter WIDTH = 8)
               (input                  clk, en,
                input      [WIDTH-1:0] d,
                output reg [WIDTH-1:0] q);

  always @(posedge clk)
    if (en) q <= d;
endmodule
```

- Memread, irwrite, addr, etc are set up just after clk edge
- Data comes back sometime after that (async)
- Data is captured in ir0 – ir3 on the next rising clk edge
- How does this change if reads are clocked?

44

mips + exmem



mips is expecting async reads

exmem has clocked reads

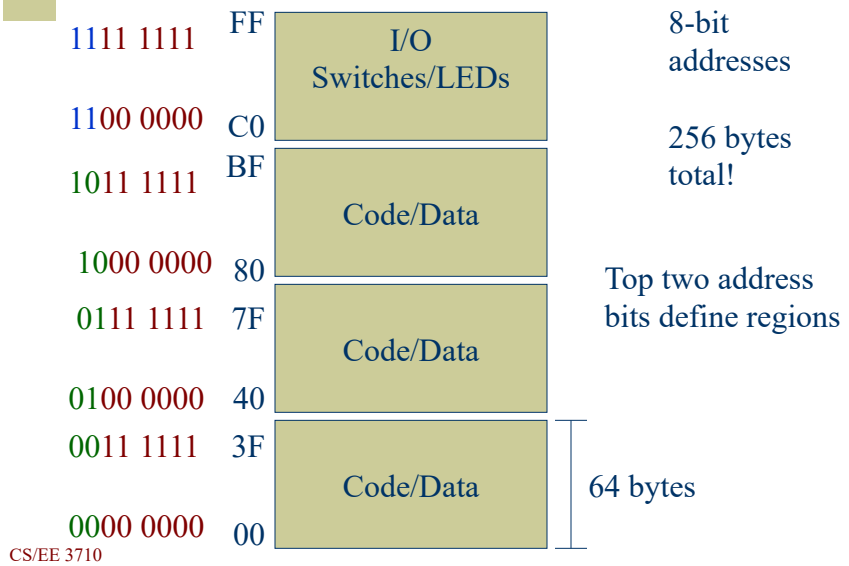
One of those rare cases where using both edges of the clock is useful!

Matches the behavior of the "two-phase clocks" described in the book…

Memory Mapped I/O

- Break memory space into pieces (ranges)
  - For some of those pieces: regular memory
  - For some of those pieces: I/O
    - That is, reading from an address in that range results in getting data from an I/O device
    - Writing to an address in that range results in data going to an I/O device

Mini-MIPS Memory Map



8-bit addresses

256 bytes total!

Top two address bits define regions

Enabled Devices

Only write to that device (i.e. enable it) if you're in the appropriate memory range.

Check top two address bits!

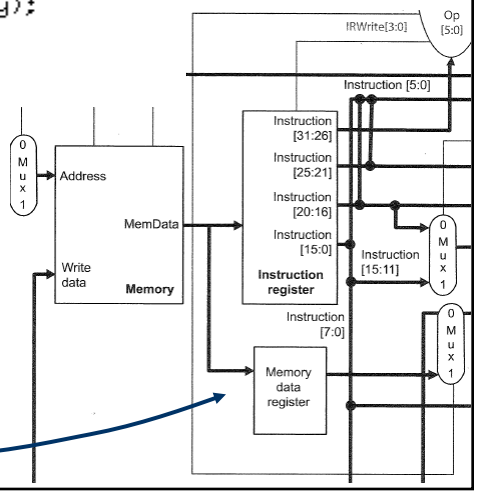
```
module flopen #(parameter WIDTH = 8)
               (input                  clk, en,
                input      [WIDTH-1:0] d,
                output reg [WIDTH-1:0] q);

   always @(posedge clk)
      if (en) q <= d;
endmodule
```

MUXes for Return Data

```
module mux2 #(parameter WIDTH = 8)
             (input  [WIDTH-1:0] d0, d1,
              input              s,
              output [WIDTH-1:0] y);

   assign y = s ? d1 : d0;
endmodule
```



Use MUX to decide if data is coming from memory or from I/O

Check address bits!

Mini-MIPS in a Nutshell

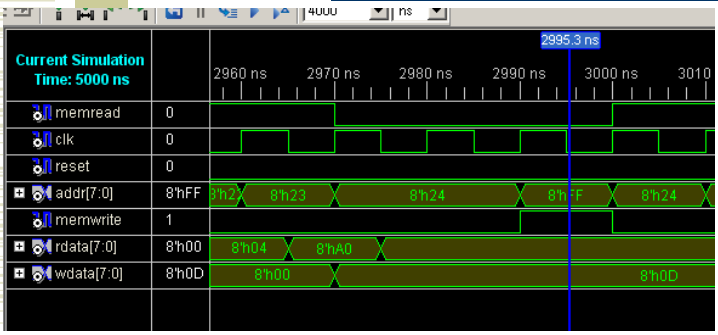
- Understand and simulate mips/exmem
  - Add ADDI instruction
  - Fibonacci program – correct if 8'0d is written to memory location 255
- Augment the system
  - Add memory mapped I/O to switches/LEDs
    - Or add simple mux switch for data return
  - Write new Fibonacci program
  - Simulate in ModelSim
  - Demonstrate on your board

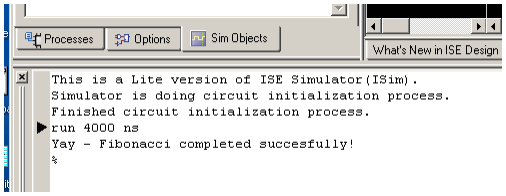
My Initial Testbench...

```
2
3  `timescale 1ns / 1ps
4
5  module mips_mem_mips_mem_sch_tb();
6
7  // Inputs
8     reg clk;
9     reg reset;
10
11 // Output
12    wire memread;
13
14 // Bidirs
15
16 // Instantiate the UUT
17    mips_mem UUT (
18        .clk(clk),
19        .memread(memread),
20        .reset(reset)
21    );
22 // Initialize Inputs
23        initial begin
24        reset <= 1;
25        #22 reset <= 0;
26        end
27
28        // Generate clock to sequence tests
29        always
30        begin
31           clk <= 1; # 5; clk <= 0; # 5;
32        end
33
34        // check the data on the memory interface between mips and exmem
35        // If you're writing, and the address is 255, then the data should
36        // be 8'h0d if you've computed the correct 8th Fibonacci number
37        always@(negedge clk)
38        if (UUT.memwrite)
39           if (UUT.addr == 8'd255 & UUT.wdata == 8'h0d)
40              $display("Yay - Fibonacci completed succesfully!");
41              else $display("Oops - wrong value written to addr 255: %h", UUT.wdata);
42
```


51

My Initial Results





52