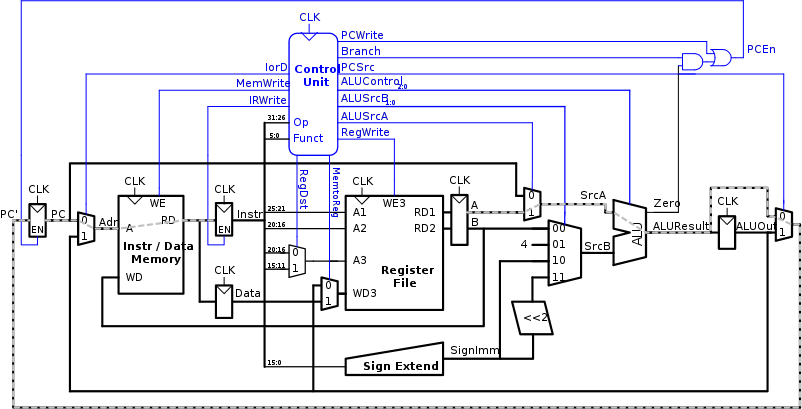
|  |
| --- |
|  |
| Multicycle Processor |
| 412 Final Project |
|  |
| **Aaron Donawerth, Kevin Phan, Lucas Magasweran** |
| **5/4/2011** |



|  |
| --- |
|  |

Contents

[Multicycle Implementation 4](#_Toc292238090)

[Multicycle Datapath 5](#_Toc292238091)

[R-Type 7](#_Toc292238092)

[I-Type 7](#_Toc292238093)

[J-Type 8](#_Toc292238094)

[Performance Analysis 9](#_Toc292238095)

[Verilog 11](#_Toc292238096)

[mipstest.v 11](#_Toc292238097)

[topmulti.v 12](#_Toc292238098)

[mipsmulti.v 12](#_Toc292238099)

[mipsparts.v 17](#_Toc292238100)

[mipsmem.v 19](#_Toc292238101)

[Mipstest.asm 19](#_Toc292238102)

[Simulation 22](#_Toc292238103)

[Bibliography 23](#_Toc292238104)

[Reference of Images 24](#_Toc292238105)

# Multicycle Implementation

The multicycle implementation fixes many of the shortcomings of the single cycle design. The single cycle processor must have a clock period long enough to support the slowest instruction. However the multicycle is not hampered by this limitation. With multicycle, multiple clock cycles with shorter periods are used. Economy of hardware is another weakness of single cycle implementations. Multicycle implementations reuses components of the datapath, making it more cost efficient than the single cycle processor.

Multiple clock cycles allows the processor to break up an instruction into multiple shorter steps. With multicycle, a subset of actions required for one instruction is performed in one cycle. This allows shorter instructions to be executed faster. This process is analogous to a dental office allotting time to patients in multiples of 15 minutes, depending on the amount of work that is anticipated[[1]](#footnote-1). Figure 1 illustrates the single cycle and multicycle clock periods and how clock period affects instruction execution.

Clock

Clock

Instr 2

Instr 1

Instr 3

Instr 4

3 cycles

3 cycles

4 cycles

5 cycles

Time

saved

Instr 1

Instr 4

Instr 3

Instr 2

Time

needed

Time

needed

Time

allotted

Time

allotted

Figure : Multicycle vs Single Cycle

The economy of hardware is addresses by reusing components or combining them. The single cycle uses three adders (two for PC logic and one ALU) and separate memory for data and instructions. The multicycle implementation combines the data and instruction memory and uses one ALU to execute all addition instructions. Adders are expensive circuits because they take up space and require extra transistors.

The design of the multicycle processor is similar to the single cycle. The multicycle processor consists of the datapath and controller block. A controller is added to produce different signals on different steps during execution. These two portions are connected to an external memory. Inside the datapath, combinational logic connects architectural state elements. Non architectural state elements such as registers are used to hold intermediate results between stages.

# Multicycle Datapath

The multicycle datapath builds upon the single cycle. The multicycle datapath has a PC register and a Register File similar to the single cycle datapath. Unlike the single cycle, the multicycle combines the data and instruction memories. Other components such as multiplexers, sign extenders, and ALU are included in the datapath. The full datapath is shown in the figure below.

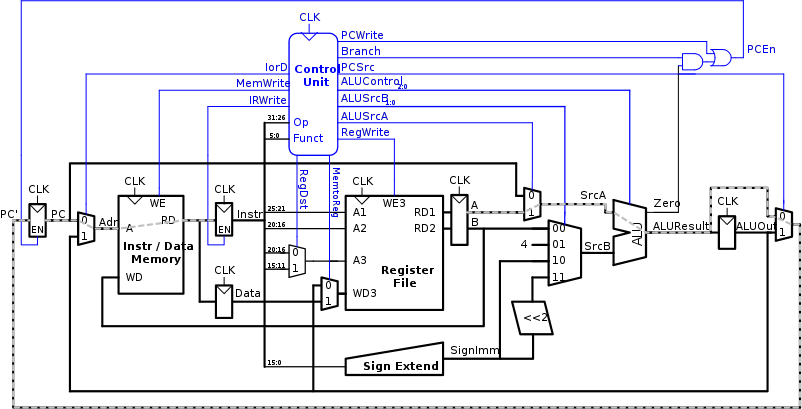


Figure : Multicycle Datapath

The multicycle datapath follows the five stage execution process: Fetch, decode, ALU, data access, and register write. Below are the control signals for the fetch and decode steps.

|  |
| --- |
| IodD = 0 |
| AluSrcA = 0 |
| ALUSrcB = 01 |
| ALUOp = 00 |
| PCSrc = 0 |
| IRWrite = 1 |
| PCWrite = 1 |

Figure 3: Fetch Control Signals

|  |
| --- |
| ALUSrcA = 0 |
| ALUSrcB = 11 |
| ALUOp = 00 |

Figure 4: Decode Control Signals

The fetch control signals are used to calculate PC+4 for the next instruction. The decode control signals are mainly used for branching. After the fetch and decode portions, the datapath of I-type, J-type, and R-type instructions vary.

## R-Type

If the opcode is an R type instruction, the result must be calculated using the ALU and stored back to the register. To carry out ALU calculation, ALUSrcA is set to 1, ALUSrcB is set to 0, and ALUOp is set to 10. ALUSrcA selects the $rs register to be used as SrcA and ALUSrcB selects $rt register to be used as SrcB of the ALU. ALUOp set to 10 indicates to the controller that ALU operation mode is dependent on the function field of the instruction. For result storage, RegDst and RegWrite are set to 1 and MemtoReg is set to 0. RegDst selects $rd register as the write destination and MemtoReg indicates the data to be written is from ALU. RegWrite serves as a write enable for the Register File.

## I-Type

Unlike the R type instruction, not all I-type instructions are carried out the same. The load word (lw), store word (sw), add immediate (addi), and branch if equal (beq) use different amounts of cycles. After the lw and sw instructions are decoded, the address for memory access must be computed by adding a base address located in the $rs register and a sign extended immediate. Control signals must be set to control the multiplexers that handle inputs at various sections of the datapath. The appropriate control signals for this step are ALUSrcA to 1, ALUSrcB to 10, and ALUOp to 00. Next, the calculated address is used to access memory; IorD is set to 1 to indicate that the incoming address is from the ALU. For sw, MemWrite is set to 1. The data located in the WriteData (WD) portion is stored to memory. Register $rt is always fed to WD, however the data from $rt is not written to memory unless MemWrite is asserted. The sw instruction is done, but lw has to write back to the register. Three control signals are set: RegDst to 0, MemtoReg to 1, and RegWrite to 1. Similar to the memory portion, if RegWrite is not set to 1, the data inside WD3 will not be written to the Register File.

For addi, $rs is still added to a sign extended immediate, but instead of using the result to access memory, the result is stored in the address located in $rt. The control signals for the ALU computation remain the same as the lw and sw instructions. Afterwards, the result is written to the Register File. The control signals for this are RegDst and RegWrite to 1 and MemtoReg to 0.

The beq instruction is has less stages than all of the other I type instructions. The branch is evaluated immediately when the ALU result is calculated. To test if a branch is equal, the values stored in $rs and $rt are subtracted from each other. If the differences between the two registers are zero, the values are equal. The result of the subtraction is indicated by the zero signal from the ALU. Once the zero signal is set to one, the result is fed into a two input AND gate with the branch signal. The result of that AND gate will produce a 1 and make PCEn 1. While this takes place, the result of the ALU is fed into the PC. With PCEn set to 1, the value of PC will be overwritten with this result. The control signals needed to carry out the branch are ALUSrcA to 1, ALUSrcB to 00, ALUOp to 01, Branch to 1, and PCSrc to 1.

## J-Type

Whenever a J-Type instruction is indicated by the opcode, the 26 least significant bits are taken from the instruction and modified as a pseudo direct address. After the instruction is decoded, the PCSrc control signal is set to 10 and the PCWrite to 1. The PCSrc signal allows the ALUResult to circumvent the register and go directly to the PC. The PCWrite makes the OR gate of PCEn to produced a 1 and enable the PC register to be overwritten.

To further understand how these datapaths are linked together, a state diagram of the Finite State Machine (FSM) is used. The state diagram is also helpful in gauging the performance of the multicycle processor. The FSM is shown below.

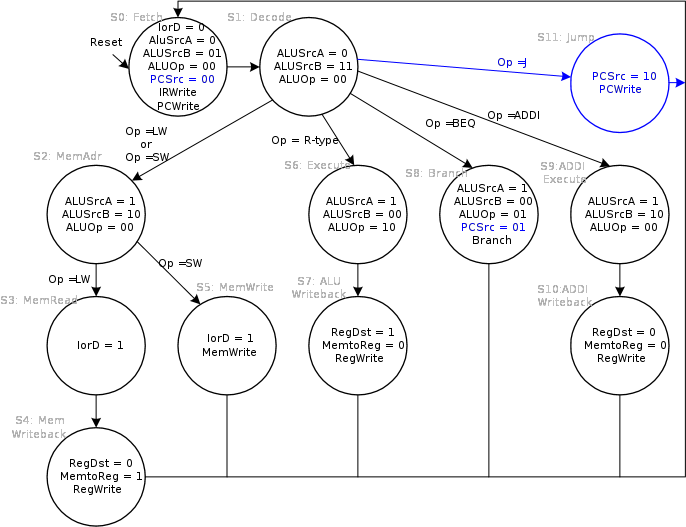


Figure : FSM of Multicycle

# Performance Analysis

The number of cycles and cycle time determine the instruction execution time. Even though the single cycle only uses one cycle, the multicycle does less work per cycle. The number of cycles per multicycle instruction is equivalent to the number of stages per instruction. Below is the number of cycles needed for each instruction.

|  |
| --- |
| Load Word = 5 Cycles |
| Store Word = 4 Cycles |
| R –Type = 4 Cycles |
| Branch = 3 Cycles |
| Jump = 3 Cycles |

Figure 5: Required Cycles for Instructions

The cycles per instruction (CPI) of the multicycle processor can be calculated by taking weighted averages of the types of instructions. The SPECINT2000 benchmark lists the instruction distribution as 25% loads, 10% stores, 11% branches, 2% jumps, and 52% R-type instructions.

|  |
| --- |
|  |

Figure6: CPI Calculation

The CPI of the multicycle is better than the worst case CPI of 5. The single cycle performance is hindered by the worst instruction. The CPI can be used to determine the MIPS of a processor and the execution time. The MIPS of a processor is calculated by dividing the frequency of the processor by its CPI. Execution time is calculated by the following equation.

# Verilog

## mipstest.v

// HDL Example 7.12 MIPS TESTBENCH

// Test bench for MIPS processor

module testbench();

reg clk;

reg reset;

wire [31:0] writedata, dataadr;

wire memwrite;

// keep track of execution status

reg [31:0] cycle;

// instantiate device to be tested

topmulti dut(clk, reset, writedata, dataadr, memwrite);

// initialize test

initial

begin

reset <= 1; # 12; reset <= 0;

cycle <= 1;

end

// generate clock to sequence tests

always

begin

clk <= 1; # 5; clk <= 0; # 5;

cycle <= cycle + 1;

end

// check results

// If successful, it should write the value 7 to address 84

always@(negedge clk)

begin

if (memwrite) begin

if (dataadr === 84 & writedata == 7) begin

$display("Simulation succeeded");

$stop;

end else if (dataadr !== 80) begin

$display("Simulation failed");

$stop;

end

end

end

endmodule

## topmulti.v

// Top-level Module of a Multicycle MIPS processor

// From Exercise 7.22

module topmulti(input clk, reset,

output [31:0] writedata, adr,

output memwrite);

wire [31:0] readdata;

// instantiate processor and memory

mips mips(clk, reset, adr, writedata, memwrite, readdata);

mem mem(clk, memwrite, adr, writedata, readdata);

endmodule

## mipsmulti.v

// Multicycle MIPS processor

module mips(input clk, reset,

output [31:0] adr, writedata,

output memwrite,

input [31:0] readdata);

wire zero, pcen, irwrite, regwrite,

alusrca, iord, memtoreg, regdst;

wire [1:0] alusrcb;

wire [1:0] pcsrc;

wire [2:0] alucontrol;

wire [5:0] op, funct;

// The control unit receives the current instruction from the datapath and tells the

// datapath how to execute that instruction.

controller c(clk, reset, op, funct, zero,

pcen, memwrite, irwrite, regwrite,

alusrca, iord, memtoreg, regdst,

alusrcb, pcsrc, alucontrol);

// The datapath operates on words of data. It

// contains structures such as memories, registers, ALUs, and multiplexers.

// MIPS is a 32-bit architecture, so we will use a 32-bit datapath.

datapath dp(clk, reset,

pcen, irwrite, regwrite,

alusrca, iord, memtoreg, regdst,

alusrcb, pcsrc, alucontrol,

op, funct, zero,

adr, writedata, readdata);

endmodule

// The main controller produces multiplexer select and register enable

// signals for the datapath. The select signals are MemtoReg, RegDst,

// IorD, PCSrc, ALUSrcB, and ALUSrcA. The enable signals are IRWrite,

// MemWrite, PCWrite, Branch, and RegWrite.

module controller(input clk, reset,

input [5:0] op, funct,

input zero,

output pcen, memwrite, irwrite, regwrite,

output alusrca, iord, memtoreg, regdst,

output [1:0] alusrcb,

output [1:0] pcsrc,

output [2:0] alucontrol);

wire [1:0] aluop;

wire branch, pcwrite;

// Main Decoder and ALU Decoder subunits.

maindec md(clk, reset, op,

pcwrite, memwrite, irwrite, regwrite,

alusrca, branch, iord, memtoreg, regdst,

alusrcb, pcsrc, aluop);

aludec ad(funct, aluop, alucontrol);

assign pcen = pcwrite | (branch & zero);

endmodule

// The controller receives the current instruction from the datapath

// and tell the datapath how to execute that instruction.

module maindec(input clk, reset,

input [5:0] op,

output pcwrite, memwrite, irwrite, regwrite,

output alusrca, branch, iord, memtoreg, regdst,

output [1:0] alusrcb,

output [1:0] pcsrc,

output [1:0] aluop);

// FSM States

parameter FETCH = 5'b00000; // State 0

parameter DECODE = 5'b00001; // State 1

parameter MEMADR = 5'b00010; // State 2

parameter MEMRD = 5'b00011; // State 3

parameter MEMWB = 5'b00100; // State 4

parameter MEMWR = 5'b00101; // State 5

parameter EXECUTE = 5'b00110; // State 6

parameter ALUWRITEBACK = 5'b00111; // State 7

parameter BRANCH = 5'b01000; // State 8

parameter ADDIEXECUTE = 5'b01001; // State 9

parameter ADDIWRITEBACK = 5'b01010; // state a

parameter JUMP = 5'b01011; // State b

// MIPS Instruction Opcodes

parameter LW = 6'b100011; // load word lw

parameter SW = 6'b101011; // store word sw

parameter RTYPE = 6'b000000; // R-type

parameter BEQ = 6'b000100; // branch if equal beq

parameter ADDI = 6'b001000; // add immidiate addi

parameter J = 6'b000010; // jump j

reg [4:0] state, nextstate;

reg [16:0] controls;

// state register

always @(posedge clk or posedge reset)

if(reset) state <= FETCH;

else state <= nextstate;

// next state logic

always @( \* )

case(state)

FETCH: nextstate <= DECODE;

DECODE: case(op)

LW: nextstate <= MEMADR;

SW: nextstate <= MEMADR;

RTYPE: nextstate <= EXECUTE;

BEQ: nextstate <= BRANCH;

ADDI: nextstate <= ADDIEXECUTE;

J: nextstate <= JUMP;

default: nextstate <= FETCH; // should never happen

endcase

MEMADR: case(op)

LW: nextstate <= MEMRD;

SW: nextstate <= MEMWR;

default: nextstate <= FETCH; // should never happen

endcase

MEMRD: nextstate <= MEMWB;

MEMWB: nextstate <= FETCH;

MEMWR: nextstate <= FETCH;

EXECUTE: nextstate <= ALUWRITEBACK;

ALUWRITEBACK: nextstate <= FETCH;

BRANCH: nextstate <= FETCH;

ADDIEXECUTE: nextstate <= ADDIWRITEBACK;

ADDIWRITEBACK: nextstate <= FETCH;

JUMP: nextstate <= FETCH;

default: nextstate <= FETCH; // should never happen

endcase

// output logic

assign {pcwrite, memwrite, irwrite, regwrite,

alusrca, branch, iord, memtoreg, regdst,

alusrcb, pcsrc,

aluop} = controls;

always @( \* )

case(state)

FETCH: controls <= 19'b1010\_00000\_0100\_00;

DECODE: controls <= 19'b0000\_00000\_1100\_00;

MEMADR: controls <= 19'b0000\_10000\_1000\_00;

MEMRD: controls <= 19'b0000\_00100\_0000\_00;

MEMWB: controls <= 19'b0001\_00010\_0000\_00;

MEMWR: controls <= 19'b0100\_00100\_0000\_00;

EXECUTE: controls <= 19'b0000\_10000\_0000\_10;

ALUWRITEBACK: controls <= 19'b0001\_00001\_0000\_00;

BRANCH: controls <= 19'b0000\_11000\_0001\_01;

ADDIEXECUTE: controls <= 19'b0000\_10000\_1000\_00;

ADDIWRITEBACK: controls <= 19'b0001\_00000\_0000\_00;

JUMP: controls <= 19'b1000\_00000\_0010\_00;

default: controls <= 19'b0000\_xxxxx\_xxxx\_xx; // should never happen

endcase

endmodule

module aludec(input [5:0] funct,

input [1:0] aluop,

output reg [2:0] alucontrol);

always @( \* )

case(aluop)

3'b000: alucontrol <= 3'b010; // add

3'b001: alucontrol <= 3'b010; // sub

// RTYPE instruction use the 6-bit funct field of instruction to specify ALU operation

3'b010: case(funct)

6'b100000: alucontrol <= 3'b010; // ADD

6'b100010: alucontrol <= 3'b110; // SUB

6'b100100: alucontrol <= 3'b000; // AND

6'b100101: alucontrol <= 3'b001; // OR

6'b101010: alucontrol <= 3'b111; // SLT

default: alucontrol <= 3'bxxx; // ???

endcase

default: alucontrol <= 3'bxxx; // ???

endcase

endmodule

module datapath(input clk, reset,

input pcen, irwrite, regwrite,

input alusrca, iord, memtoreg, regdst,

input [1:0] alusrcb,

input [1:0] pcsrc,

input [2:0] alucontrol,

output [5:0] op, funct,

output zero,

output [31:0] adr, writedata,

input [31:0] readdata);

// Internal signals of the datapath module

wire [4:0] writereg;

wire [31:0] pcnext, pc;

wire [31:0] instr, data, srca, srcb;

wire [31:0] a;

wire [31:0] aluresult, aluout;

wire [31:0] signimm; // the sign-extended immediate

wire [31:0] signimmsh; // the sign-extended immediate shifted left by 2

wire [31:0] wd3, rd1, rd2;

// op and funct fields to controller

assign op = instr[31:26];

assign funct = instr[5:0];

// datapath

flopenr #(32) pcreg(clk, reset, pcen, pcnext, pc);

mux2 #(32) adrmux(pc, aluout, iord, adr);

flopenr #(32) instrreg(clk, reset, irwrite, readdata, instr);

flopr #(32) datareg(clk, reset, readdata, data);

mux2 #(5) regdstmux(instr[20:16], instr[15:11], regdst, writereg);

mux2 #(32) wdmux(aluout, data, memtoreg, wd3);

regfile rf(clk, regwrite, instr[25:21], instr[20:16],

writereg, wd3, rd1, rd2);

signext se(instr[15:0], signimm);

sl2 immsh(signimm, signimmsh);

flopr #(32) areg(clk, reset, rd1, a);

flopr #(32) breg(clk, reset, rd2, writedata);

mux2 #(32) srcamux(pc, a, alusrca, srca);

mux4 #(32) srcbmux(writedata, 32'b100, signimm, signimmsh,

alusrcb, srcb);

alu alu(srca, srcb, alucontrol,

aluresult, zero);

flopr #(32) alureg(clk, reset, aluresult, aluout);

mux3 #(32) pcmux(aluresult, aluout,

{pc[31:28], instr[25:0], 2'b00}, pcsrc, pcnext);

endmodule

## mipsparts.v

// 32-bit ALU

// Function codes are defined on page 243

module alu( input [31:0] A, B,

input [2:0] F,

output reg [31:0] Y, output Zero);

always @ ( \* )

case (F[2:0])

3'b000: Y <= A & B;

3'b001: Y <= A | B;

3'b010: Y <= A + B;

//3'b011: Y <= 0; // not used

3'b011: Y <= A & ~B;

3'b101: Y <= A + ~B;

3'b110: Y <= A - B;

3'b111: Y <= A < B ? 1:0;

default: Y <= 0; //default to 0, should not happen

endcase

assign Zero = (Y == 32'b0);

endmodule

// Example 7.6 Register file

module regfile(input clk,

input we3,

input [4:0] ra1, ra2, wa3,

input [31:0] wd3,

output [31:0] rd1, rd2);

reg [31:0] rf[31:0];

// 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 (we3) rf[wa3] <= wd3;

assign rd1 = (ra1 != 0) ? rf[ra1] : 0;

assign rd2 = (ra2 != 0) ? rf[ra2] : 0;

endmodule

// Example 7.8 Left Shift (Multiply by 4)

module sl2(input [31:0] a,

output [31:0] y);

// shift left by 2

assign y = {a[29:0], 2'b00};

endmodule

// Example 7.9 Sign Extension

module signext(input [15:0] a,

output [31:0] y);

assign y = {{16{a[15]}}, a};

endmodule

// Example 7.10 Resettable Flip-flop with width parameter

module flopr #(parameter WIDTH = 8)

(input clk, reset,

input [WIDTH-1:0] d,

output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else q <= d;

endmodule

// Example 4.20 RESETTABLE ENABLED REGISTER with width parameter

module flopenr #(parameter WIDTH = 8)

(input clk, reset,

input en,

input [WIDTH-1:0] d,

output reg [WIDTH-1:0] q);

always @(posedge clk, posedge reset)

if (reset) q <= 0;

else if (en) q <= d;

endmodule

// Example 4.5 2:1 MULTIPLEXER with width parameter

module mux2 #(parameter WIDTH = 8)

(input [WIDTH-1:0] d0, d1,

input s,

output [WIDTH-1:0] y);

assign y = s ? d1 : d0;

endmodule

// 3:1 MULTIPLEXER with width parameter

module mux3 #(parameter WIDTH = 8)

(input [WIDTH-1:0] d0, d1, d2,

input [1:0] s,

output [WIDTH-1:0] y);

assign #1 y = s[1] ? d2 : (s[0] ? d1 : d0);

endmodule

// Example 4.6 4:1 MULTIPLEXER with width parameter

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

## mipsmem.v

// Multicycle MIPS instruction and data memory

// "memfile.dat" contains a test program

module mem(input clk, we,

input [31:0] a, wd,

output [31:0] rd);

reg [31:0] RAM[63:0];

initial

begin

$readmemh("memfile.dat",RAM);

end

assign rd = RAM[a[31:2]]; // word aligned

always @(posedge clk)

if (we)

RAM[a[31:2]] <= wd;

Endmodule

## Mipstest.asm

main: addi $2, $0, # initialize $2 = 5 0 20020005

addi $3, $0, 12 # initialize $3 = 12 4 2003000c

addi $7, $3, -9 # initialize $7 = 3 8 2067fff7

or $4, $7, $2 # $4 <= 3 or 5 = 7 c 00e22025

and $5, $3, $4 # $5 <= 12 and 7 = 4 10 00642824

add $5, $5, $4 # $5 = 4 + 7 = 11 14 00a42820

beq $5, $7, end # shouldn’t be taken 18 10a7000a

slt $4, $3, $4 # $4 = 12 < 7 = 0 1c 0064202a

beq $4, $0, around # should be taken 20 10800001

addi $5, $0, 0 # shouldn’t happen 24 20050000

around: slt $4, $7, $2 # $4 = 3 < 5 = 1 28 00e2202a

add $7, $4, $5 # $7 = 1 + 11 = 12 2c 00853820

sub $7, $7, $2 # $7 = 12 - 5 = 7 30 00e23822

sw $7, 68($3) # [80] = 7 34 ac670044

lw $2, 80($0) # $2 = [80] = 7 38 8c020050

j end # should be taken 3c 08000011

addi $2, $0, 1 # shouldn’t happen 40 20020001

end: sw $2, 84($0) # write adr 84 = 7 44 ac020054

The verilog code is based off of the following images:

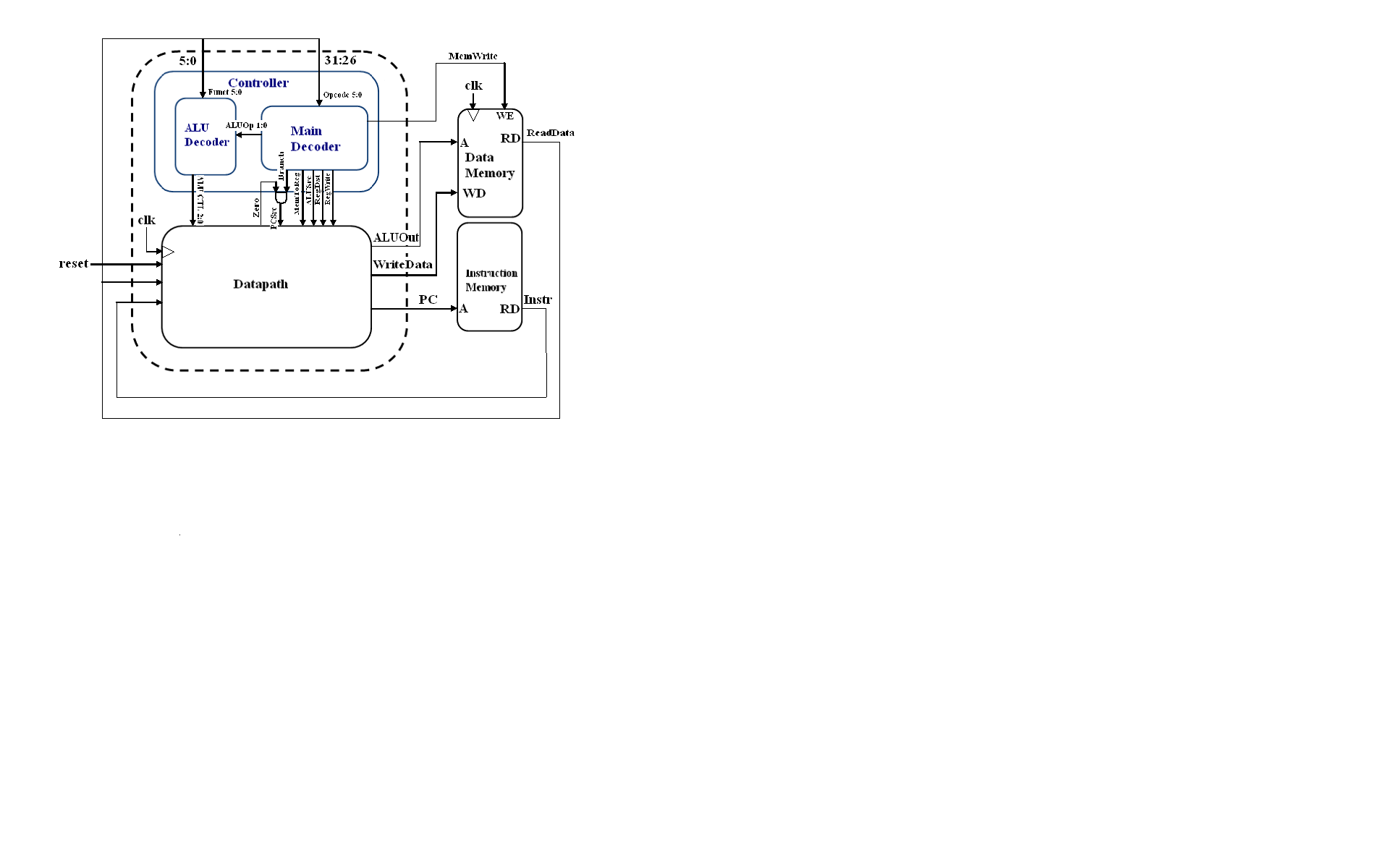


Figure 7: Single Cycle MIPS Processor

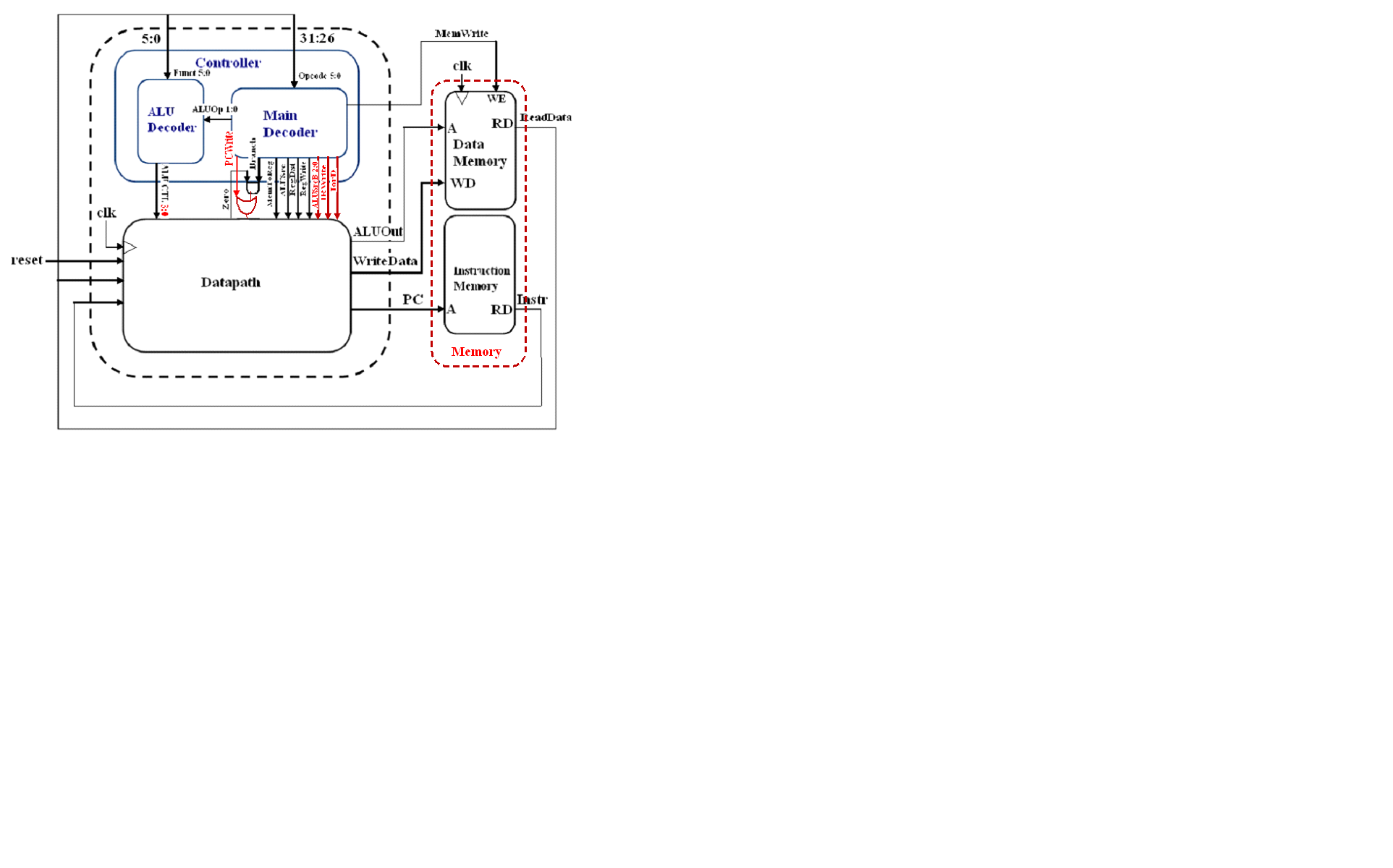


Figure 8: Multicycle MIPS Processor

# Simulation

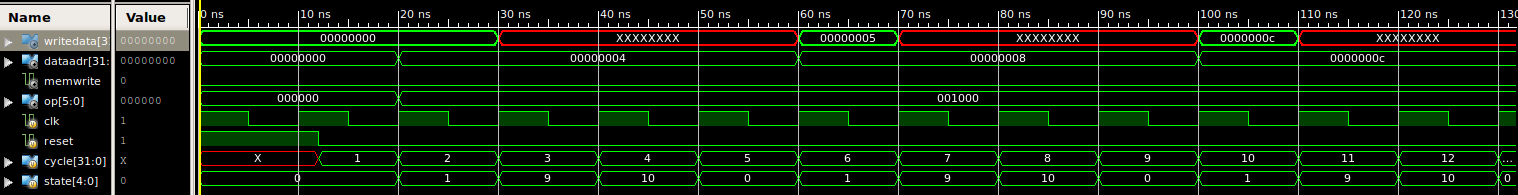


Figure 7: Simulation 0-70 ns

The Simulation starts by asserting the “reset” signal to force the controller back to state 0 and clear any random data that appears during the turn on process. At 10 ns, the processor starts its first state 0 operation or the fetch stage, where the instruction is retrieved from memory according to the Program Counter or PC. Once the Instruction is fetched, it then enters the second stat or the decode state. At this point, the opcode appears and the command we were using was addi. At this point the program differs depending on the opcode and would enter the appropriate state according to the op code and continue to execute according to that path. For addi that involves state 9 and 10, at 30 and 40 ns respectively. Once these states are completed the controller would return to state 0 to fetch the next instruction. However the data still has not been written back to the register designated in the addi instruction. In looking at figure 2 you can see that the data is stored in a clocked register during stage 10. When the Controller enters state 0 again, the rising clock also triggers the register and puts it onto the write data line of the register block. One clock later the data is then written to the register during the 60 ns period. This shows that the actual length of the addi instruction is 4 cycles like expected, however it also showed that the data may not be ready within that same time frame due to the clocking and registers that allow the multicycle MIPS processor to operate with the least amount of hardware.

# Bibliography

Harris, David Money. Harris, Sarah L. Digital Design and Computer Architecture, San Francisco: Morgan Kaufman Publishers, 2007

Parhami, Behrooz. Computer Architecture, New York: Oxford University Press, 2005

# Reference of Images

Figure 1: Parhami, Behrooz. Computer Architecture

Figure 2: Harris, David Money. Harris, Sarah L. Digital Design and Computer Architecture

Figure 5: Harris, David Money. Harris, Sarah L. Digital Design and Computer Architecture

1. Computer Architecture – Behrooz Parhami [↑](#footnote-ref-1)