fkamal000@citymail.cuny.edu

[Email address]

Faheem Kamal

Professor Gertner

Cs343 Lab 5 Report

Single Cycle CPU-Lite

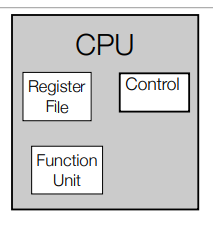
Objective

The goal of this assignment was to develop the Single Cycle CPU is a processor that carries out one instruction in a single clock cycle. It is based on the 32-bit MIPS Instruction Set Architecture.

Introduction

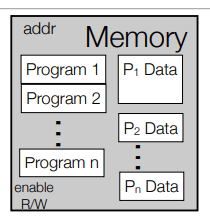
MIPS (Microprocessor without Interlocked Pipelined Stages) is a Reduced Instruction Set Computer (RISC) instruction set architecture. All operations are of the form Rd <- Rs op Rt. MIPS are “load-store” architectures which means that all operations are performed only on operands in registers. This also means that the only instructions that can access memory are load and stores. MIPS was originally a research project at Stanford under John Hennessy and was eventually commercialized by MIPS Technologies and now owns a huge share of the market in embedded space.

Parts of the CPU:



* Register File – Current Data being operated upon
* Function Unit – Combinational Logic that Does the Computation
* Control – Keeps Track of the Current Program Instruction

Memory:

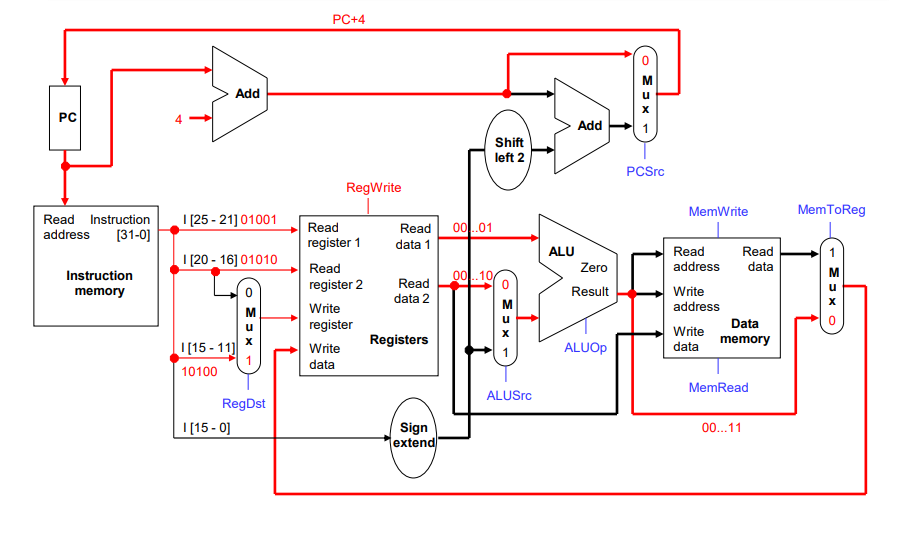


* It serves as a big storage tank.
* Stores programs that needs to be/being executed.
* Data.
* Special Structures such as Heap or Stack.

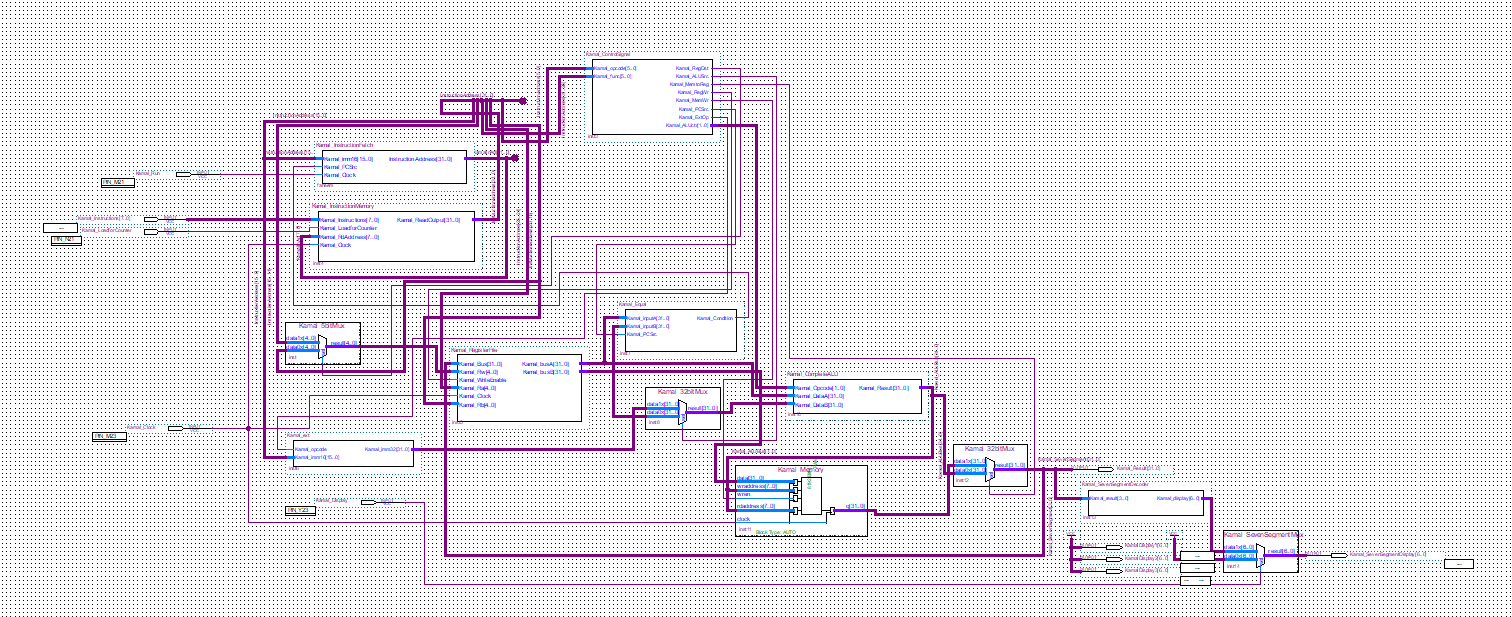
ISA

* An Instruction Set Architecture is an interface between the hardware and software.
* Consists of:
  + A set of Operations (Instructions)
  + Data Units (32-bit Data Word)
  + Processor State (Registers) – 32-bit Registers
  + Input and Output Control – Load and Store
  + Execution Model – 32-bit Program Counter

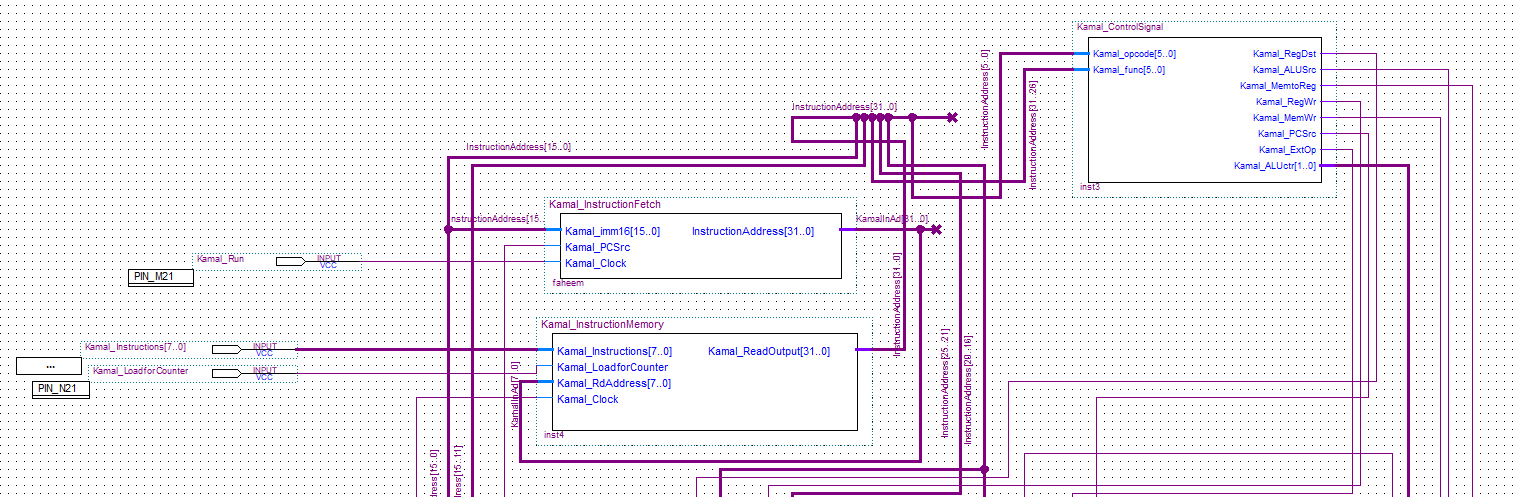
For my design, I will discuss the components in my Single Cycle CPU.



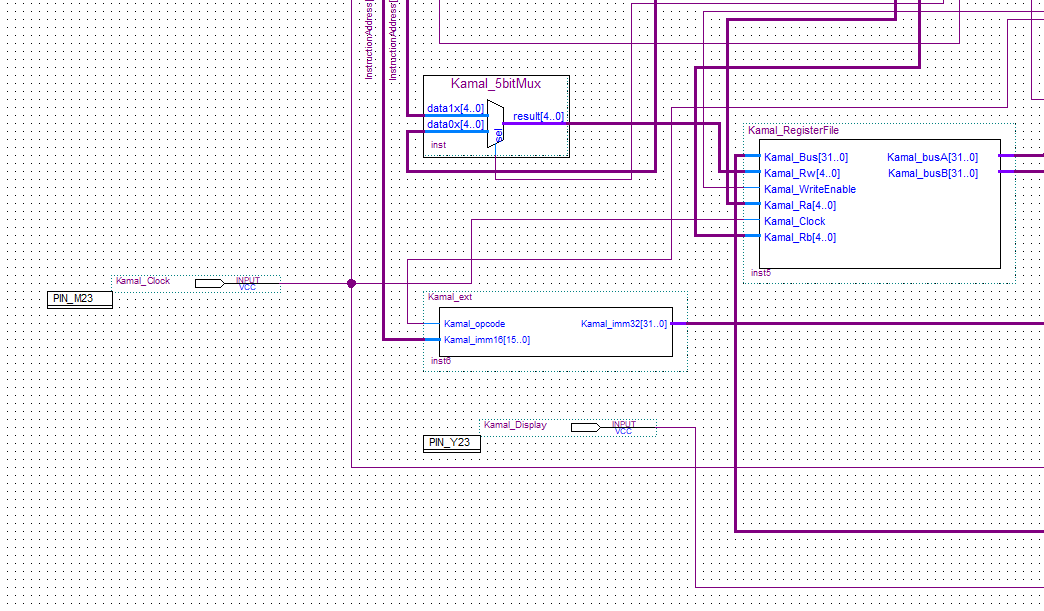
Schematic of MIPS Single Cycle CPU



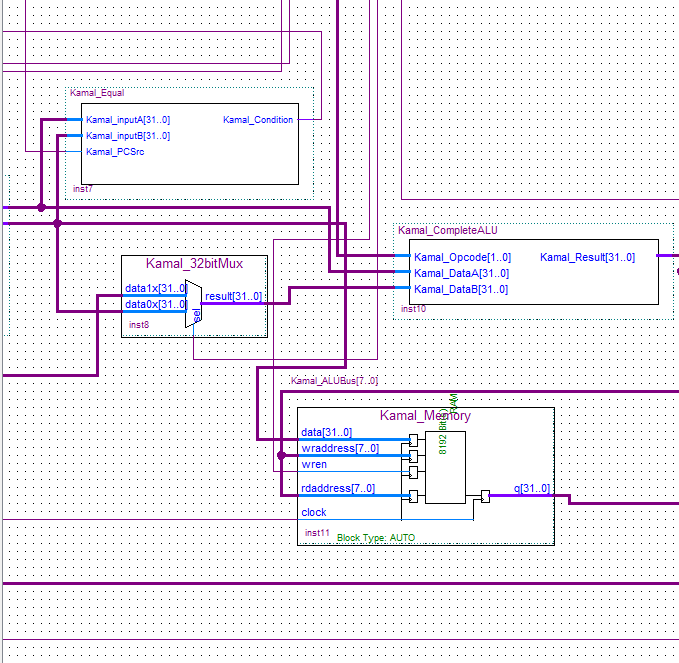
Schematic of MIPS Single Cycle CPU in Quartus II (Full-View)



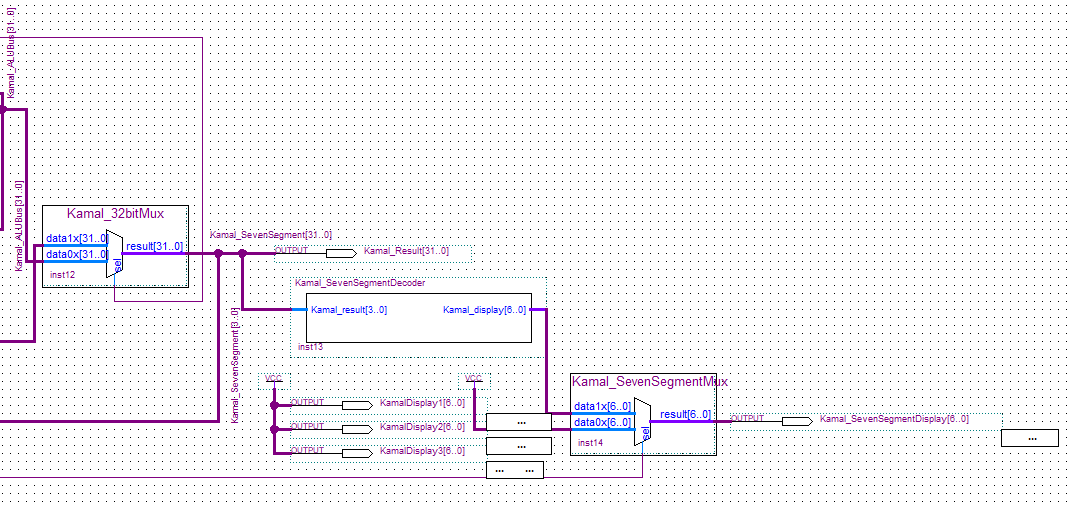
Schematic of MIPS Single Cycle CPU in Quartus II (Top-View)



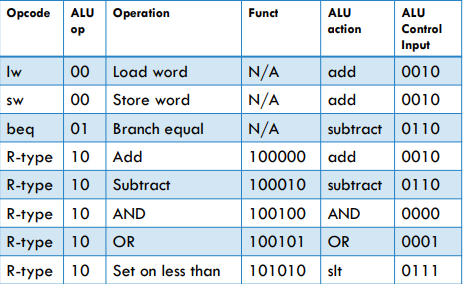
Schematic of MIPS Single Cycle CPU in Quartus II (Bottom Left-View)



Schematic of MIPS Single Cycle CPU in Quartus II (Bottom Middle-View)



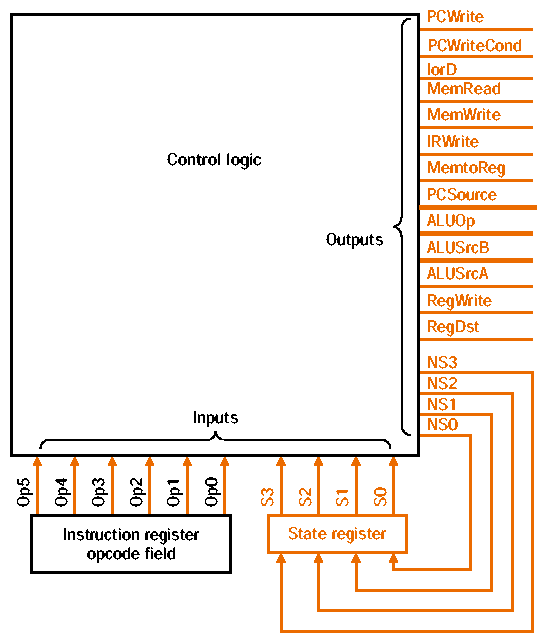
Schematic of MIPS Single Cycle CPU in Quartus II (Bottom Right-View)



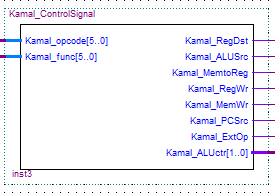
Set of ALU Opcodes

Single-Cycle CPU Parts:

1. MIPS Control (Control)



MIPS Control Unit Schematic



MIPS Control Unit in Quartus II

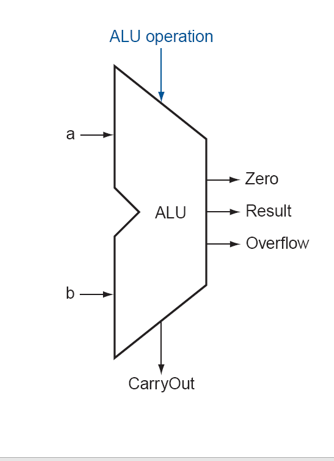
The MIPS Control Unit decodes Instructions to determine what segments will be active on the Datapath. It generates signals to:

* Set Muxes to Correct Input – Designed 32-Bit and 5-Bit Muxes
* Operation Code to ALU – Designed ALU
* Read and Write to Register File – Designed Register File
* Read and Write to Memory (Load/Store) – Designed Memory
* Update of Program Counter (Branches) – Designed Counter
* Branch Target Address Computation

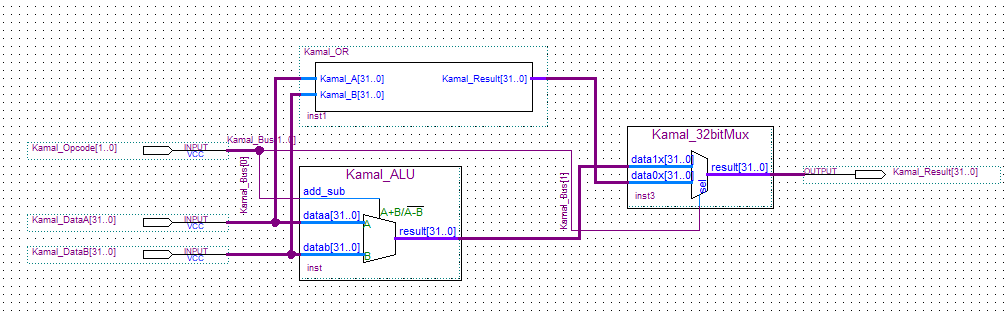
For the Control Signal:

* Kamal\_RegDst – Which Field for Write Register
  + Determines how the destination register is specified (rt or rd in Patterson and Hennessey).
* Kamal\_ALUSrc – Source for second ALU input
  + Selects the second source operand for the ALU (rt or sign-extended immediate field in Patterson and Hennessey).
* Kamal\_MemtoReg – Source of Write Register port data input
  + Determines where the value to be written comes from (ALU result or memory in Patterson and Hennessey).
* Kamal\_RegWr – Write to Register File
  + Enables a write to one of the registers.
* Kamal\_MemWr – Write Input Address/Data to Memory
  + Enables a memory write for store instructions.
* Kamal\_PCSrc – Source for PC (PC + 4 or other Target Address)
* Kamal\_ExtOp – Zero/Signed Extension

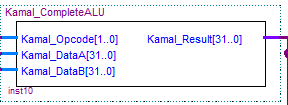
1. ALU (Function Unit)



Schematic of the ALU



ALU Schematic in Quartus II

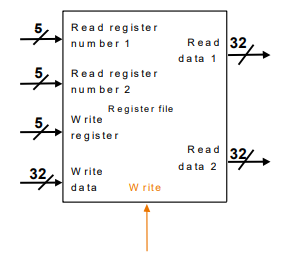


Complete ALU in Quartus II

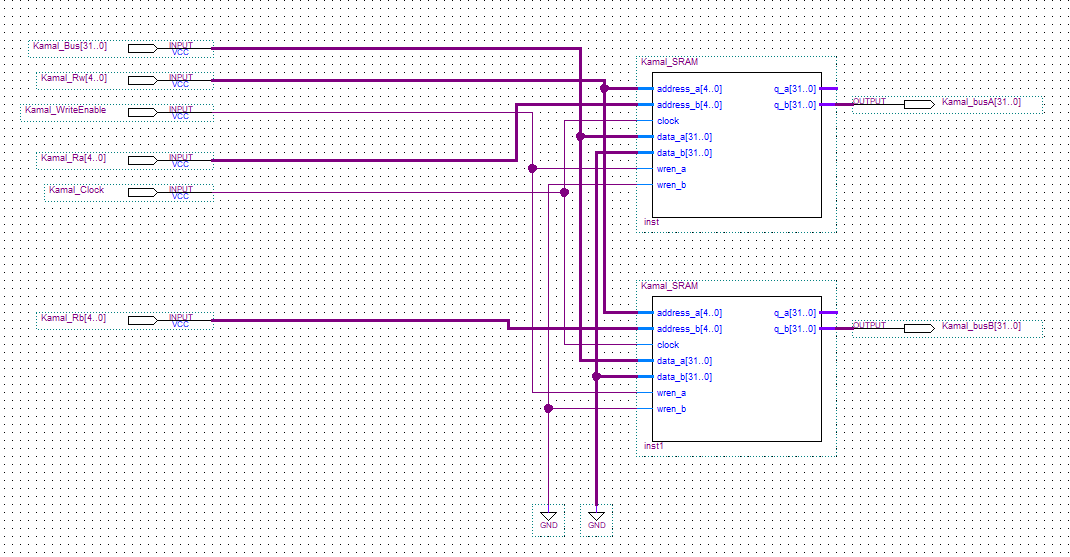
The ALU (Arithmetic Logic Unit) is a combinational circuit capable of computing a variety of arithmetic and logical functions. In MIPS, the ALU takes 2 32-bit Inputs and produces 32-bit Output, plus some additional signals.

* Operations needed for MIPS instructions discussed so far: add, subtact, and, or, zero test, comparison [MIPS also has nor, xor; multiply and divide]
* Different circuits combined by multiplexer; Multiplexer select becomes function select for ALU
* Feed Output of High-order bit of Adder to Lower-order bit for “Set on Less” operation
* Use OR Gate on ALU output for Equal Test (Bitwise OR VHDL code provided and Equal VHDL code also provided).

1. Register File (Made up of SRAM)



Schematic of Register File



Register File in Quartus II

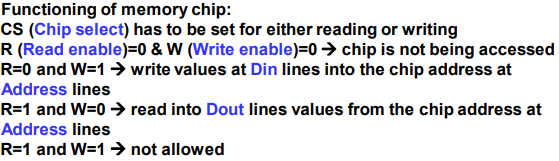
The MIPS Register File includes 32 32-bit General Purpose Registers and has 5-bit and 32-bit MUX. The Register File makes possible to simultaneously read from two registers and write into one register as it is appropriate for MIPS processor. (Code for 5-Bit Mux, 32-bit Mux and SRAM is provided in Appendix.)

A register file functions as follows:

* Any value provided on the 5-line Read register number 1 port results in the content of the corresponding register being provided on the 32-line Read data 1 port.
* Any value provided on 5-line Read register number 2 port results in the content of the corresponding register being provided on the 32-line Read data 2 port.
* on the falling edge of write line, values that appear on 32-bit Write data port are written into the register with the number specified on the 5-line Write register port.

The MIPS Register File here is built on 2 SRAM (VHDL code provided in Appendix) which is Static Random Access Memory. In SRAM technology, three-state D-Latch is a basic building block, i.e basic memory cell. Internally, the D-Latch can have a state corresponding to 0 or 1.

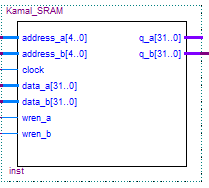
Functions of SRAM:



Furthermore, a Register File is not related to Disk Files. Instead it is a small set of high-speed storage cells inside the CPU. There are special-purpose registers such as the IR and PC, and general-purpose registers for storing operands of instructions such as add, sub, mul, etc. A CPU register can generally be accessed in a single clock cycle, whereas main memory may require dozens of CPU clock cycles to read or write. Since there are very few registers compared to memory cells, registers also require far fewer bits to specify which register to use. This in turn allows for smaller instruction codes.

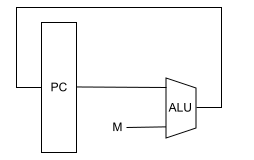
Smaller Components used to make the Single Cycle CPU (In-Depth Analysis)

1. Register File – SRAM

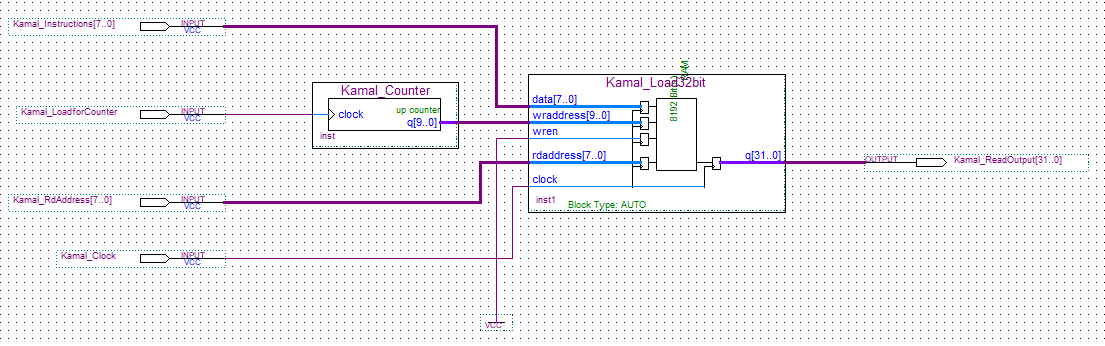


SRAM in Quartus II

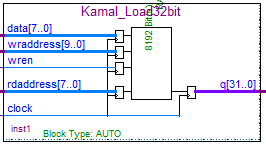
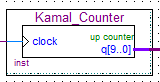
1. Instruction Memory – Program Counter, Load 32-bit MUX



Schematic of Program Counter



Full View of Instruction Memory in Quartus II

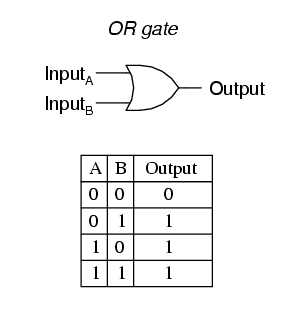


Program Counter and Load 32-Bit MUX in Quartus II

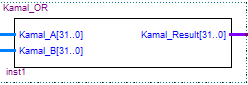
The Program Counter (PC) is a register structure that contains the address pointer value of the current instruction. Each cycle, the value at the pointer is read into the instruction decoder and the program counter is updated to point to the next instruction. For RISC computers updating the PC register is as simple as adding the machine word length (in bytes) to the PC. In a CISC machine, however, the length of the current instruction needs to be calculated, and that length value needs to be added to the PC.

The Load 32-Bit MUX serves as a way for instructions to be loaded into Memory and will then be stored for further execution.

1. ALU – Bitwise OR, 32-Bit MUX

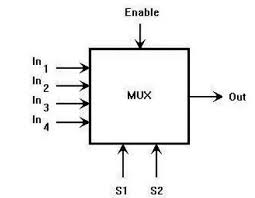


General Schematic and Truth Table of OR Gate

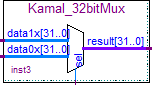


Schematic of 32-Bit Bitwise OR in Quartus II

The OR gate is a digital logic gate that implements logical disjunction – it behaves according to the truth table to the right. A HIGH output (1) results if one or both the inputs to the gate are HIGH (1). If neither input is high, a LOW output (0) results. In another sense, the function of OR effectively finds the maximum between two binary digits, just as the complementary AND function finds the minimum.



Schematic of MUX

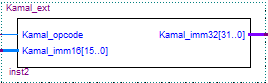


Schematic of 32-Bit MUX in Quartus II

In electronics, a multiplexer (or mux) is a device that selects between several analog or digital input signals and forwards it to a single output line. A multiplexer of 2^n select lines, which are used to select which input line to send to the output. Multiplexers are mainly used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth. A multiplexer is also called a data selector. Multiplexers can also be used to implement Boolean functions of multiple variables.

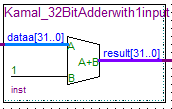
An electronic multiplexer makes it possible for several signals to share one device or resource, for example, one A/D converter or one communication line, instead of having one device per input signal.

1. Instruction Fetch – Zero/Signed Extension, 32 Bit Adder with 1 Input, 32 Bit Adder, 32 Bit Mux and Instruction Register



Zero/Signed Extension in Quartus II

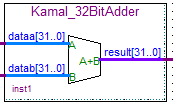
An integer register on the MIPS is 32 bits. When a value is loaded from memory with fewer than 32 bits, the remaining bits must be assigned. Sign extension is used for signed loads of bytes (8 bits using the lb instruction) and halfwords (16 bits using the lh instruction). Sign extension replicates the most significant bit loaded into the remaining bits. Zero extension is used for unsigned loads of bytes (lbu) and halfwords (lhu). Zeroes are filled in the remaining bits.



32 Bit Adder with 1 Input in Quartus II

Each full adder inputs a Cin, which is the Cout of the previous adder. This kind of adder is called a ripple-carry adder (RCA), since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder (under the assumption that Cin = 0).

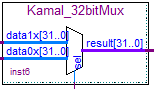
The layout of a ripple-carry adder is simple, which allows fast design time; however, the ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic. In a 32-bit ripple-carry adder, there are 32 full adders, so the critical path (worst case) delay is 3 (from input to carry in first adder) + 31 × 2 (for carry propagation in latter adders) = 65 gate delays.



32 Bit Adder in Quartus II

Each full adder inputs a Cin, which is the Cout of the previous adder. This kind of adder is called a ripple-carry adder (RCA), since each carry bit "ripples" to the next full adder. Note that the first (and only the first) full adder may be replaced by a half adder (under the assumption that Cin = 0).

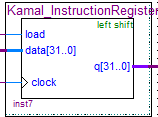
The layout of a ripple-carry adder is simple, which allows fast design time; however, the ripple-carry adder is relatively slow, since each full adder must wait for the carry bit to be calculated from the previous full adder. The gate delay can easily be calculated by inspection of the full adder circuit. Each full adder requires three levels of logic. In a 32-bit ripple-carry adder, there are 32 full adders, so the critical path (worst case) delay is 3 (from input to carry in first adder) + 31 × 2 (for carry propagation in latter adders) = 65 gate delays.



32 Bit Mux in Quartus II

In electronics, a multiplexer (or mux) is a device that selects between several analog or digital input signals and forwards it to a single output line. A multiplexer of 2^n select lines, which are used to select which input line to send to the output. Multiplexers are mainly used to increase the amount of data that can be sent over the network within a certain amount of time and bandwidth. A multiplexer is also called a data selector. Multiplexers can also be used to implement Boolean functions of multiple variables.

An electronic multiplexer makes it possible for several signals to share one device or resource, for example, one A/D converter or one communication line, instead of having one device per input signal.



32 Bit Instruction Register in Quartus II

CONCLUSION

The single-cycle datapath is not used in modern processors, because it is inefficient. The critical path (longest propagation sequence through the datapath) is five components for the load instruction. The cycle time tc is limited by the settling time ts of these components. For a circuit with no feedback loops, tc > 5ts. In practice, tc = 5kts, with large proportionality constant k, due to feedback loops, delayed settling due to circuit noise, etc. It is possible to compute the required execution time for each instruction class from the critical path information. The result is that the Load instruction takes 5 units of time, while the Store and R-format instructions take 4 units of time. All the other types of instructions that the datapath is designed to execute run faster, requiring three units of time.

The problem of penalizing addition, subtraction, and comparison operations to accommodate loads and stores leads one to ask if multiple cycles of a much faster clock could be used for each part of the fetch-decode-execute cycle. In practice, this technique is employed in CPU design and implementation, as discussed in the following sections on multicycle datapath design. In Section 5, we will show that datapath actions can be *interleaved* in time to yield a potentially fast implementation of the fetch-decode-execute cycle that is formalized in a technique called *pipelining*.

APPENDIX – VHDL CODE

1. Sign/Zero Extension

library ieee;

use ieee.std\_logic\_1164.all;

-- Zero/ Sign Extend

entity Kamal\_ext is

port(

Kamal\_opcode : in std\_logic;

Kamal\_imm16 : in std\_logic\_vector (15 downto 0);

Kamal\_imm32 : out std\_logic\_vector (31 downto 0)

);

end Kamal\_ext;

architecture behave of Kamal\_ext is

begin

Kamal\_imm32 <= ("1111111111111111" & Kamal\_imm16) when ((Kamal\_opcode = '1') and (Kamal\_imm16(15) = '1')) else

("0000000000000000" & Kamal\_imm16);

end behave;

1. Seven Segment MUX

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY lpm;

USE lpm.lpm\_components.all;

ENTITY Kamal\_SevenSegmentMux IS

PORT

(

data0x : IN STD\_LOGIC\_VECTOR (6 DOWNTO 0);

data1x : IN STD\_LOGIC\_VECTOR (6 DOWNTO 0);

sel : IN STD\_LOGIC ;

result : OUT STD\_LOGIC\_VECTOR (6 DOWNTO 0)

);

END Kamal\_SevenSegmentMux;

ARCHITECTURE SYN OF kamal\_sevensegmentmux IS

-- type STD\_LOGIC\_2D is array (NATURAL RANGE <>, NATURAL RANGE <>) of STD\_LOGIC;

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (6 DOWNTO 0);

SIGNAL sub\_wire1 : STD\_LOGIC\_VECTOR (6 DOWNTO 0);

SIGNAL sub\_wire2 : STD\_LOGIC\_2D (1 DOWNTO 0, 6 DOWNTO 0);

SIGNAL sub\_wire3 : STD\_LOGIC\_VECTOR (6 DOWNTO 0);

SIGNAL sub\_wire4 : STD\_LOGIC ;

SIGNAL sub\_wire5 : STD\_LOGIC\_VECTOR (0 DOWNTO 0);

BEGIN

sub\_wire3 <= data0x(6 DOWNTO 0);

result <= sub\_wire0(6 DOWNTO 0);

sub\_wire1 <= data1x(6 DOWNTO 0);

sub\_wire2(1, 0) <= sub\_wire1(0);

sub\_wire2(1, 1) <= sub\_wire1(1);

sub\_wire2(1, 2) <= sub\_wire1(2);

sub\_wire2(1, 3) <= sub\_wire1(3);

sub\_wire2(1, 4) <= sub\_wire1(4);

sub\_wire2(1, 5) <= sub\_wire1(5);

sub\_wire2(1, 6) <= sub\_wire1(6);

sub\_wire2(0, 0) <= sub\_wire3(0);

sub\_wire2(0, 1) <= sub\_wire3(1);

sub\_wire2(0, 2) <= sub\_wire3(2);

sub\_wire2(0, 3) <= sub\_wire3(3);

sub\_wire2(0, 4) <= sub\_wire3(4);

sub\_wire2(0, 5) <= sub\_wire3(5);

sub\_wire2(0, 6) <= sub\_wire3(6);

sub\_wire4 <= sel;

sub\_wire5(0) <= sub\_wire4;

LPM\_MUX\_component : LPM\_MUX

GENERIC MAP (

lpm\_size => 2,

lpm\_type => "LPM\_MUX",

lpm\_width => 7,

lpm\_widths => 1

)

PORT MAP (

data => sub\_wire2,

sel => sub\_wire5,

result => sub\_wire0

);

END SYN;

1. 32-Bit MUX

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY lpm;

USE lpm.lpm\_components.all;

ENTITY Kamal\_32bitMux IS

PORT

(

data0x : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

data1x : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

sel : IN STD\_LOGIC ;

result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END Kamal\_32bitMux;

ARCHITECTURE SYN OF kamal\_32bitmux IS

-- type STD\_LOGIC\_2D is array (NATURAL RANGE <>, NATURAL RANGE <>) of STD\_LOGIC;

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

SIGNAL sub\_wire1 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

SIGNAL sub\_wire2 : STD\_LOGIC\_2D (1 DOWNTO 0, 31 DOWNTO 0);

SIGNAL sub\_wire3 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

SIGNAL sub\_wire4 : STD\_LOGIC ;

SIGNAL sub\_wire5 : STD\_LOGIC\_VECTOR (0 DOWNTO 0);

BEGIN

sub\_wire3 <= data0x(31 DOWNTO 0);

result <= sub\_wire0(31 DOWNTO 0);

sub\_wire1 <= data1x(31 DOWNTO 0);

sub\_wire2(1, 0) <= sub\_wire1(0);

sub\_wire2(1, 1) <= sub\_wire1(1);

sub\_wire2(1, 2) <= sub\_wire1(2);

sub\_wire2(1, 3) <= sub\_wire1(3);

sub\_wire2(1, 4) <= sub\_wire1(4);

sub\_wire2(1, 5) <= sub\_wire1(5);

sub\_wire2(1, 6) <= sub\_wire1(6);

sub\_wire2(1, 7) <= sub\_wire1(7);

sub\_wire2(1, 8) <= sub\_wire1(8);

sub\_wire2(1, 9) <= sub\_wire1(9);

sub\_wire2(1, 10) <= sub\_wire1(10);

sub\_wire2(1, 11) <= sub\_wire1(11);

sub\_wire2(1, 12) <= sub\_wire1(12);

sub\_wire2(1, 13) <= sub\_wire1(13);

sub\_wire2(1, 14) <= sub\_wire1(14);

sub\_wire2(1, 15) <= sub\_wire1(15);

sub\_wire2(1, 16) <= sub\_wire1(16);

sub\_wire2(1, 17) <= sub\_wire1(17);

sub\_wire2(1, 18) <= sub\_wire1(18);

sub\_wire2(1, 19) <= sub\_wire1(19);

sub\_wire2(1, 20) <= sub\_wire1(20);

sub\_wire2(1, 21) <= sub\_wire1(21);

sub\_wire2(1, 22) <= sub\_wire1(22);

sub\_wire2(1, 23) <= sub\_wire1(23);

sub\_wire2(1, 24) <= sub\_wire1(24);

sub\_wire2(1, 25) <= sub\_wire1(25);

sub\_wire2(1, 26) <= sub\_wire1(26);

sub\_wire2(1, 27) <= sub\_wire1(27);

sub\_wire2(1, 28) <= sub\_wire1(28);

sub\_wire2(1, 29) <= sub\_wire1(29);

sub\_wire2(1, 30) <= sub\_wire1(30);

sub\_wire2(1, 31) <= sub\_wire1(31);

sub\_wire2(0, 0) <= sub\_wire3(0);

sub\_wire2(0, 1) <= sub\_wire3(1);

sub\_wire2(0, 2) <= sub\_wire3(2);

sub\_wire2(0, 3) <= sub\_wire3(3);

sub\_wire2(0, 4) <= sub\_wire3(4);

sub\_wire2(0, 5) <= sub\_wire3(5);

sub\_wire2(0, 6) <= sub\_wire3(6);

sub\_wire2(0, 7) <= sub\_wire3(7);

sub\_wire2(0, 8) <= sub\_wire3(8);

sub\_wire2(0, 9) <= sub\_wire3(9);

sub\_wire2(0, 10) <= sub\_wire3(10);

sub\_wire2(0, 11) <= sub\_wire3(11);

sub\_wire2(0, 12) <= sub\_wire3(12);

sub\_wire2(0, 13) <= sub\_wire3(13);

sub\_wire2(0, 14) <= sub\_wire3(14);

sub\_wire2(0, 15) <= sub\_wire3(15);

sub\_wire2(0, 16) <= sub\_wire3(16);

sub\_wire2(0, 17) <= sub\_wire3(17);

sub\_wire2(0, 18) <= sub\_wire3(18);

sub\_wire2(0, 19) <= sub\_wire3(19);

sub\_wire2(0, 20) <= sub\_wire3(20);

sub\_wire2(0, 21) <= sub\_wire3(21);

sub\_wire2(0, 22) <= sub\_wire3(22);

sub\_wire2(0, 23) <= sub\_wire3(23);

sub\_wire2(0, 24) <= sub\_wire3(24);

sub\_wire2(0, 25) <= sub\_wire3(25);

sub\_wire2(0, 26) <= sub\_wire3(26);

sub\_wire2(0, 27) <= sub\_wire3(27);

sub\_wire2(0, 28) <= sub\_wire3(28);

sub\_wire2(0, 29) <= sub\_wire3(29);

sub\_wire2(0, 30) <= sub\_wire3(30);

sub\_wire2(0, 31) <= sub\_wire3(31);

sub\_wire4 <= sel;

sub\_wire5(0) <= sub\_wire4;

LPM\_MUX\_component : LPM\_MUX

GENERIC MAP (

lpm\_size => 2,

lpm\_type => "LPM\_MUX",

lpm\_width => 32,

lpm\_widths => 1

)

PORT MAP (

data => sub\_wire2,

sel => sub\_wire5,

result => sub\_wire0

);

END SYN;

1. 5-Bit MUX

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY lpm;

USE lpm.lpm\_components.all;

ENTITY Kamal\_5bitMux IS

PORT

(

data0x : IN STD\_LOGIC\_VECTOR (4 DOWNTO 0);

data1x : IN STD\_LOGIC\_VECTOR (4 DOWNTO 0);

sel : IN STD\_LOGIC ;

result : OUT STD\_LOGIC\_VECTOR (4 DOWNTO 0)

);

END Kamal\_5bitMux;

ARCHITECTURE SYN OF kamal\_5bitmux IS

-- type STD\_LOGIC\_2D is array (NATURAL RANGE <>, NATURAL RANGE <>) of STD\_LOGIC;

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (4 DOWNTO 0);

SIGNAL sub\_wire1 : STD\_LOGIC\_VECTOR (4 DOWNTO 0);

SIGNAL sub\_wire2 : STD\_LOGIC\_2D (1 DOWNTO 0, 4 DOWNTO 0);

SIGNAL sub\_wire3 : STD\_LOGIC\_VECTOR (4 DOWNTO 0);

SIGNAL sub\_wire4 : STD\_LOGIC ;

SIGNAL sub\_wire5 : STD\_LOGIC\_VECTOR (0 DOWNTO 0);

BEGIN

sub\_wire3 <= data0x(4 DOWNTO 0);

result <= sub\_wire0(4 DOWNTO 0);

sub\_wire1 <= data1x(4 DOWNTO 0);

sub\_wire2(1, 0) <= sub\_wire1(0);

sub\_wire2(1, 1) <= sub\_wire1(1);

sub\_wire2(1, 2) <= sub\_wire1(2);

sub\_wire2(1, 3) <= sub\_wire1(3);

sub\_wire2(1, 4) <= sub\_wire1(4);

sub\_wire2(0, 0) <= sub\_wire3(0);

sub\_wire2(0, 1) <= sub\_wire3(1);

sub\_wire2(0, 2) <= sub\_wire3(2);

sub\_wire2(0, 3) <= sub\_wire3(3);

sub\_wire2(0, 4) <= sub\_wire3(4);

sub\_wire4 <= sel;

sub\_wire5(0) <= sub\_wire4;

LPM\_MUX\_component : LPM\_MUX

GENERIC MAP (

lpm\_size => 2,

lpm\_type => "LPM\_MUX",

lpm\_width => 5,

lpm\_widths => 1

)

PORT MAP (

data => sub\_wire2,

sel => sub\_wire5,

result => sub\_wire0

);

END SYN;

1. Instruction Register

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY lpm;

USE lpm.all;

ENTITY Kamal\_InstructionRegister IS

PORT

(

clock : IN STD\_LOGIC ;

data : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

load : IN STD\_LOGIC ;

q : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END Kamal\_InstructionRegister;

ARCHITECTURE SYN OF kamal\_instructionregister IS

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

COMPONENT lpm\_shiftreg

GENERIC (

lpm\_direction : STRING;

lpm\_type : STRING;

lpm\_width : NATURAL

);

PORT (

clock : IN STD\_LOGIC ;

data : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

load : IN STD\_LOGIC ;

q : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END COMPONENT;

BEGIN

q <= sub\_wire0(31 DOWNTO 0);

LPM\_SHIFTREG\_component : LPM\_SHIFTREG

GENERIC MAP (

lpm\_direction => "LEFT",

lpm\_type => "LPM\_SHIFTREG",

lpm\_width => 32

)

PORT MAP (

clock => clock,

data => data,

load => load,

q => sub\_wire0

);

END SYN;

1. Counter

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY lpm;

USE lpm.all;

ENTITY Kamal\_Counter IS

PORT

(

clock : IN STD\_LOGIC ;

q : OUT STD\_LOGIC\_VECTOR (9 DOWNTO 0)

);

END Kamal\_Counter;

ARCHITECTURE SYN OF kamal\_counter IS

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (9 DOWNTO 0);

COMPONENT lpm\_counter

GENERIC (

lpm\_direction : STRING;

lpm\_port\_updown : STRING;

lpm\_type : STRING;

lpm\_width : NATURAL

);

PORT (

clock : IN STD\_LOGIC ;

q : OUT STD\_LOGIC\_VECTOR (9 DOWNTO 0)

);

END COMPONENT;

BEGIN

q <= sub\_wire0(9 DOWNTO 0);

LPM\_COUNTER\_component : LPM\_COUNTER

GENERIC MAP (

lpm\_direction => "UP",

lpm\_port\_updown => "PORT\_UNUSED",

lpm\_type => "LPM\_COUNTER",

lpm\_width => 10

)

PORT MAP (

clock => clock,

q => sub\_wire0

);

END SYN;

1. Memory

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY altera\_mf;

USE altera\_mf.all;

ENTITY Kamal\_Memory IS

PORT

(

clock : IN STD\_LOGIC := '1';

data : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

rdaddress : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);

wraddress : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);

wren : IN STD\_LOGIC := '0';

q : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END Kamal\_Memory;

ARCHITECTURE SYN OF kamal\_memory IS

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

COMPONENT altsyncram

GENERIC (

address\_aclr\_b : STRING;

address\_reg\_b : STRING;

clock\_enable\_input\_a : STRING;

clock\_enable\_input\_b : STRING;

clock\_enable\_output\_b : STRING;

intended\_device\_family : STRING;

lpm\_type : STRING;

numwords\_a : NATURAL;

numwords\_b : NATURAL;

operation\_mode : STRING;

outdata\_aclr\_b : STRING;

outdata\_reg\_b : STRING;

power\_up\_uninitialized : STRING;

read\_during\_write\_mode\_mixed\_ports : STRING;

widthad\_a : NATURAL;

widthad\_b : NATURAL;

width\_a : NATURAL;

width\_b : NATURAL;

width\_byteena\_a : NATURAL

);

PORT (

address\_a : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);

clock0 : IN STD\_LOGIC ;

data\_a : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

q\_b : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0);

wren\_a : IN STD\_LOGIC ;

address\_b : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0)

);

END COMPONENT;

BEGIN

q <= sub\_wire0(31 DOWNTO 0);

altsyncram\_component : altsyncram

GENERIC MAP (

address\_aclr\_b => "NONE",

address\_reg\_b => "CLOCK0",

clock\_enable\_input\_a => "BYPASS",

clock\_enable\_input\_b => "BYPASS",

clock\_enable\_output\_b => "BYPASS",

intended\_device\_family => "Cyclone IV E",

lpm\_type => "altsyncram",

numwords\_a => 256,

numwords\_b => 256,

operation\_mode => "DUAL\_PORT",

outdata\_aclr\_b => "NONE",

outdata\_reg\_b => "CLOCK0",

power\_up\_uninitialized => "FALSE",

read\_during\_write\_mode\_mixed\_ports => "DONT\_CARE",

widthad\_a => 8,

widthad\_b => 8,

width\_a => 32,

width\_b => 32,

width\_byteena\_a => 1

)

PORT MAP (

address\_a => wraddress,

clock0 => clock,

data\_a => data,

wren\_a => wren,

address\_b => rdaddress,

q\_b => sub\_wire0

);

END SYN;

1. Seven Segment Decoder

library IEEE;

use IEEE.STD\_LOGIC\_1164.ALL;

entity Kamal\_SevenSegmentDecoder is

Port (

Kamal\_result : in STD\_LOGIC\_VECTOR (3 downto 0);

Kamal\_display : out STD\_LOGIC\_VECTOR (6 downto 0)

);

end Kamal\_SevenSegmentDecoder;

architecture behave of Kamal\_SevenSegmentDecoder is

begin

process(Kamal\_result)

begin

case Kamal\_result is

when "0000" =>

Kamal\_display <= "0000001"; -- 0

when "0001" =>

Kamal\_display <= "1001111"; -- 1

when "0010" =>

Kamal\_display <= "0010010"; -- 2

when "0011" =>

Kamal\_display <= "0000110"; -- 3

when "0100" =>

Kamal\_display <= "1001100"; -- 4

when "0101" =>

Kamal\_display <= "0100100"; -- 5

when "0110" =>

Kamal\_display <= "0100000"; -- 6

when "0111" =>

Kamal\_display <= "0001111"; -- 7

when "1000" =>

Kamal\_display <= "0000000"; -- 8

when "1001" =>

Kamal\_display <= "0000100"; -- 9

when others =>

Kamal\_display <= "1111111"; -- null

end case;

end process;

end behave;

1. Bitwise-OR

library ieee;

use ieee.std\_logic\_1164.all;

entity Kamal\_OR is

port(

Kamal\_A : in std\_logic\_vector(31 downto 0);

Kamal\_B : in std\_logic\_vector(31 downto 0);

Kamal\_Result : out std\_logic\_vector(31 downto 0)

);

end Kamal\_OR;

architecture behave of Kamal\_OR is

begin

Kamal\_Result <= Kamal\_A or Kamal\_B;

end behave;

1. ALU

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY lpm;

USE lpm.all;

ENTITY Kamal\_ALU IS

PORT

(

add\_sub : IN STD\_LOGIC ;

dataa : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

datab : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END Kamal\_ALU;

ARCHITECTURE SYN OF kamal\_alu IS

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

COMPONENT lpm\_add\_sub

GENERIC (

lpm\_direction : STRING;

lpm\_hint : STRING;

lpm\_representation : STRING;

lpm\_type : STRING;

lpm\_width : NATURAL

);

PORT (

add\_sub : IN STD\_LOGIC ;

dataa : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

datab : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END COMPONENT;

BEGIN

result <= sub\_wire0(31 DOWNTO 0);

LPM\_ADD\_SUB\_component : LPM\_ADD\_SUB

GENERIC MAP (

lpm\_direction => "UNUSED",

lpm\_hint => "ONE\_INPUT\_IS\_CONSTANT=NO,CIN\_USED=NO",

lpm\_representation => "UNSIGNED",

lpm\_type => "LPM\_ADD\_SUB",

lpm\_width => 32

)

PORT MAP (

add\_sub => add\_sub,

dataa => dataa,

datab => datab,

result => sub\_wire0

);

END SYN;

1. 32-Bit Adder

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY lpm;

USE lpm.all;

ENTITY Kamal\_32BitAdder IS

PORT

(

dataa : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

datab : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END Kamal\_32BitAdder;

ARCHITECTURE SYN OF kamal\_32bitadder IS

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

COMPONENT lpm\_add\_sub

GENERIC (

lpm\_direction : STRING;

lpm\_hint : STRING;

lpm\_representation : STRING;

lpm\_type : STRING;

lpm\_width : NATURAL

);

PORT (

dataa : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

datab : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END COMPONENT;

BEGIN

result <= sub\_wire0(31 DOWNTO 0);

LPM\_ADD\_SUB\_component : LPM\_ADD\_SUB

GENERIC MAP (

lpm\_direction => "ADD",

lpm\_hint => "ONE\_INPUT\_IS\_CONSTANT=NO,CIN\_USED=NO",

lpm\_representation => "UNSIGNED",

lpm\_type => "LPM\_ADD\_SUB",

lpm\_width => 32

)

PORT MAP (

dataa => dataa,

datab => datab,

result => sub\_wire0

);

END SYN;

1. 32-Bit Adder with 1 Input

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY lpm;

USE lpm.all;

ENTITY Kamal\_32BitAdderwith1input IS

PORT

(

dataa : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END Kamal\_32BitAdderwith1input;

ARCHITECTURE SYN OF kamal\_32bitadderwith1input IS

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

SIGNAL sub\_wire1\_bv : BIT\_VECTOR (31 DOWNTO 0);

SIGNAL sub\_wire1 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

COMPONENT lpm\_add\_sub

GENERIC (

lpm\_direction : STRING;

lpm\_hint : STRING;

lpm\_representation : STRING;

lpm\_type : STRING;

lpm\_width : NATURAL

);

PORT (

dataa : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

datab : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

result : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END COMPONENT;

BEGIN

sub\_wire1\_bv(31 DOWNTO 0) <= "00000000000000000000000000000001";

sub\_wire1 <= To\_stdlogicvector(sub\_wire1\_bv);

result <= sub\_wire0(31 DOWNTO 0);

LPM\_ADD\_SUB\_component : LPM\_ADD\_SUB

GENERIC MAP (

lpm\_direction => "ADD",

lpm\_hint => "ONE\_INPUT\_IS\_CONSTANT=YES,CIN\_USED=NO",

lpm\_representation => "UNSIGNED",

lpm\_type => "LPM\_ADD\_SUB",

lpm\_width => 32

)

PORT MAP (

dataa => dataa,

datab => sub\_wire1,

result => sub\_wire0

);

END SYN;

1. SRAM

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY altera\_mf;

USE altera\_mf.all;

ENTITY Kamal\_SRAM IS

PORT

(

address\_a : IN STD\_LOGIC\_VECTOR (4 DOWNTO 0);

address\_b : IN STD\_LOGIC\_VECTOR (4 DOWNTO 0);

clock : IN STD\_LOGIC := '1';

data\_a : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

data\_b : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

wren\_a : IN STD\_LOGIC := '0';

wren\_b : IN STD\_LOGIC := '0';

q\_a : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0);

q\_b : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END Kamal\_SRAM;

ARCHITECTURE SYN OF kamal\_sram IS

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

SIGNAL sub\_wire1 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

COMPONENT altsyncram

GENERIC (

address\_reg\_b : STRING;

clock\_enable\_input\_a : STRING;

clock\_enable\_input\_b : STRING;

clock\_enable\_output\_a : STRING;

clock\_enable\_output\_b : STRING;

indata\_reg\_b : STRING;

intended\_device\_family : STRING;

lpm\_type : STRING;

numwords\_a : NATURAL;

numwords\_b : NATURAL;

operation\_mode : STRING;

outdata\_aclr\_a : STRING;

outdata\_aclr\_b : STRING;

outdata\_reg\_a : STRING;

outdata\_reg\_b : STRING;

power\_up\_uninitialized : STRING;

read\_during\_write\_mode\_mixed\_ports : STRING;

read\_during\_write\_mode\_port\_a : STRING;

read\_during\_write\_mode\_port\_b : STRING;

widthad\_a : NATURAL;

widthad\_b : NATURAL;

width\_a : NATURAL;

width\_b : NATURAL;

width\_byteena\_a : NATURAL;

width\_byteena\_b : NATURAL;

wrcontrol\_wraddress\_reg\_b : STRING

);

PORT (

clock0 : IN STD\_LOGIC ;

wren\_a : IN STD\_LOGIC ;

address\_b : IN STD\_LOGIC\_VECTOR (4 DOWNTO 0);

data\_b : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

q\_a : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0);

wren\_b : IN STD\_LOGIC ;

address\_a : IN STD\_LOGIC\_VECTOR (4 DOWNTO 0);

data\_a : IN STD\_LOGIC\_VECTOR (31 DOWNTO 0);

q\_b : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END COMPONENT;

BEGIN

q\_a <= sub\_wire0(31 DOWNTO 0);

q\_b <= sub\_wire1(31 DOWNTO 0);

altsyncram\_component : altsyncram

GENERIC MAP (

address\_reg\_b => "CLOCK0",

clock\_enable\_input\_a => "BYPASS",

clock\_enable\_input\_b => "BYPASS",

clock\_enable\_output\_a => "BYPASS",

clock\_enable\_output\_b => "BYPASS",

indata\_reg\_b => "CLOCK0",

intended\_device\_family => "Cyclone IV E",

lpm\_type => "altsyncram",

numwords\_a => 32,

numwords\_b => 32,

operation\_mode => "BIDIR\_DUAL\_PORT",

outdata\_aclr\_a => "NONE",

outdata\_aclr\_b => "NONE",

outdata\_reg\_a => "CLOCK0",

outdata\_reg\_b => "CLOCK0",

power\_up\_uninitialized => "FALSE",

read\_during\_write\_mode\_mixed\_ports => "DONT\_CARE",

read\_during\_write\_mode\_port\_a => "NEW\_DATA\_NO\_NBE\_READ",

read\_during\_write\_mode\_port\_b => "NEW\_DATA\_NO\_NBE\_READ",

widthad\_a => 5,

widthad\_b => 5,

width\_a => 32,

width\_b => 32,

width\_byteena\_a => 1,

width\_byteena\_b => 1,

wrcontrol\_wraddress\_reg\_b => "CLOCK0"

)

PORT MAP (

clock0 => clock,

wren\_a => wren\_a,

address\_b => address\_b,

data\_b => data\_b,

wren\_b => wren\_b,

address\_a => address\_a,

data\_a => data\_a,

q\_a => sub\_wire0,

q\_b => sub\_wire1

);

END SYN;

1. MIPS Control Unit

library ieee;

use ieee.std\_logic\_1164.all;

-- Mips Control Signals --

ENTITY Kamal\_ControlSignal IS

PORT

(

Kamal\_opcode : IN STD\_LOGIC\_VECTOR (5 DOWNTO 0); -- Opcodes

Kamal\_func : IN STD\_LOGIC\_VECTOR (5 DOWNTO 0); -- Determines or Specifies Particular Arithmetic Operation -

Kamal\_RegDst : OUT STD\_LOGIC; -- Determines how the Destination Register is specified --

Kamal\_ALUSrc : OUT STD\_LOGIC; -- Controls Multiplexer to select either a register operand(0) or a constant operand(1)-

Kamal\_MemtoReg : OUT STD\_LOGIC; -- Determines where the value to be written comes from --

Kamal\_RegWr : OUT STD\_LOGIC; -- Enables a write to one of the Registers --

Kamal\_MemWr : OUT STD\_LOGIC; -- Enables a memory write to store instructions --

Kamal\_PCSrc : OUT STD\_LOGIC; -- Source for PC (PC + 4 or target address) --

Kamal\_ExtOp : OUT STD\_LOGIC; -- Zero/Sign Extension --

Kamal\_ALUctr : OUT STD\_LOGIC\_VECTOR (1 DOWNTO 0) -- Either specifies the ALU operation to be performed or specifies that the operation should be determined from the function bits --

);

END Kamal\_ControlSignal;

ARCHITECTURE arch OF Kamal\_ControlSignal IS

BEGIN

Process (Kamal\_opcode)

BEGIN

Kamal\_RegWr <= '0';

CASE Kamal\_opcode IS

When "000000" =>

Kamal\_RegDst <= '1';

Kamal\_ALUSrc <= '0';

Kamal\_MemtoReg <= '0';

Kamal\_RegWr <= '1';

Kamal\_MemWr <= '0';

Kamal\_PCSrc <= '0';

Kamal\_ExtOp <= '0';

Kamal\_ALUctr <= "00";

When "100011" =>

Kamal\_RegDst <= '0';

Kamal\_ALUSrc <= '1';

Kamal\_MemtoReg <= '1';

Kamal\_RegWr <= '1';

Kamal\_MemWr <= '0';

Kamal\_PCSrc <= '0';

Kamal\_ExtOp <= '0';

Kamal\_ALUctr <= "01";

When "101011" =>

Kamal\_RegDst <= '0';

Kamal\_ALUSrc <= '1';

Kamal\_MemtoReg <= '1';

Kamal\_RegWr <= '0';

Kamal\_MemWr <= '1';

Kamal\_PCSrc <= '0';

Kamal\_ExtOp <= '0';

Kamal\_ALUctr <= "01";

When "000100" =>

Kamal\_RegDst <= '1';

Kamal\_ALUSrc <= '0';

Kamal\_MemtoReg <= '0';

Kamal\_RegWr <= '0';

Kamal\_MemWr <= '0';

Kamal\_PCSrc <= '1';

Kamal\_ExtOp <= '0';

Kamal\_ALUctr <= "00";

When "000010" =>

Kamal\_RegDst <= '1';

Kamal\_ALUSrc <= '1';

Kamal\_MemtoReg <= '0';

Kamal\_RegWr <= '1';

Kamal\_MemWr <= '0';

Kamal\_PCSrc <= '0';

Kamal\_ExtOp <= '1';

Kamal\_ALUctr <= "01";

When OTHERS => NULL;

Kamal\_RegDst <= '0';

Kamal\_ALUSrc <= '0';

Kamal\_MemtoReg <= '0';

Kamal\_RegWr <= '0';

Kamal\_MemWr <= '0';

Kamal\_PCSrc <= '0';

Kamal\_ExtOp <= '0';

Kamal\_ALUctr <= "00";

END CASE;

END Process;

END arch;

1. Equal VHDL

library ieee;

use ieee.std\_logic\_1164.all;

entity Kamal\_Equal is

port(

Kamal\_inputA : in std\_logic\_vector(31 downto 0);

Kamal\_inputB : in std\_logic\_vector(31 downto 0);

Kamal\_PCSrc : in std\_logic;

Kamal\_Condition : out std\_logic

);

end Kamal\_Equal;

architecture behavior of Kamal\_Equal is

begin

Kamal\_Condition <= '1' when ((Kamal\_inputA = Kamal\_inputB) and (Kamal\_PCSrc = '1')) else

'0';

end behavior;

1. 32-Bit Loader

LIBRARY ieee;

USE ieee.std\_logic\_1164.all;

LIBRARY altera\_mf;

USE altera\_mf.all;

ENTITY Kamal\_Load32bit IS

PORT

(

clock : IN STD\_LOGIC := '1';

data : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);

rdaddress : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);

wraddress : IN STD\_LOGIC\_VECTOR (9 DOWNTO 0);

wren : IN STD\_LOGIC := '0';

q : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0)

);

END Kamal\_Load32bit;

ARCHITECTURE SYN OF kamal\_load32bit IS

SIGNAL sub\_wire0 : STD\_LOGIC\_VECTOR (31 DOWNTO 0);

COMPONENT altsyncram

GENERIC (

address\_aclr\_b : STRING;

address\_reg\_b : STRING;

clock\_enable\_input\_a : STRING;

clock\_enable\_input\_b : STRING;

clock\_enable\_output\_b : STRING;

intended\_device\_family : STRING;

lpm\_type : STRING;

numwords\_a : NATURAL;

numwords\_b : NATURAL;

operation\_mode : STRING;

outdata\_aclr\_b : STRING;

outdata\_reg\_b : STRING;

power\_up\_uninitialized : STRING;

read\_during\_write\_mode\_mixed\_ports : STRING;

widthad\_a : NATURAL;

widthad\_b : NATURAL;

width\_a : NATURAL;

width\_b : NATURAL;

width\_byteena\_a : NATURAL

);

PORT (

address\_a : IN STD\_LOGIC\_VECTOR (9 DOWNTO 0);

clock0 : IN STD\_LOGIC ;

data\_a : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0);

q\_b : OUT STD\_LOGIC\_VECTOR (31 DOWNTO 0);

wren\_a : IN STD\_LOGIC ;

address\_b : IN STD\_LOGIC\_VECTOR (7 DOWNTO 0)

);

END COMPONENT;

BEGIN

q <= sub\_wire0(31 DOWNTO 0);

altsyncram\_component : altsyncram

GENERIC MAP (

address\_aclr\_b => "NONE",

address\_reg\_b => "CLOCK0",

clock\_enable\_input\_a => "BYPASS",

clock\_enable\_input\_b => "BYPASS",

clock\_enable\_output\_b => "BYPASS",

intended\_device\_family => "Cyclone IV E",

lpm\_type => "altsyncram",

numwords\_a => 1024,

numwords\_b => 256,

operation\_mode => "DUAL\_PORT",

outdata\_aclr\_b => "NONE",

outdata\_reg\_b => "CLOCK0",

power\_up\_uninitialized => "FALSE",

read\_during\_write\_mode\_mixed\_ports => "DONT\_CARE",

widthad\_a => 10,

widthad\_b => 8,

width\_a => 8,

width\_b => 32,

width\_byteena\_a => 1

)

PORT MAP (

address\_a => wraddress,

clock0 => clock,

data\_a => data,

wren\_a => wren,

address\_b => rdaddress,

q\_b => sub\_wire0

);

END SYN;

1. PIN ASSIGNMENTS

To, Location

Kamal\_Instructions[7], PIN\_AB26

Kamal\_Instructions[6], PIN\_AD26

Kamal\_Instructions[5], PIN\_AC26

Kamal\_Instructions[4], PIN\_AB27

Kamal\_Instructions[3], PIN\_AD27

Kamal\_Instructions[2], PIN\_AC27

Kamal\_Instructions[1], PIN\_AC28

Kamal\_Instructions[0], PIN\_AB28

Kamal\_LoadforCounter, PIN\_N21

Kamal\_Run, PIN\_M21

Kamal\_Display, PIN\_Y23

Kamal\_Clock, PIN\_M23

Kamal\_SevenSegmentDisplay[6], PIN\_G18

Kamal\_SevenSegmentDisplay[5], PIN\_F22

Kamal\_SevenSegmentDisplay[4], PIN\_E17

Kamal\_SevenSegmentDisplay[3], PIN\_L26

Kamal\_SevenSegmentDisplay[2], PIN\_L25

Kamal\_SevenSegmentDisplay[1], PIN\_J22

Kamal\_SevenSegmentDisplay[0], PIN\_H22

KamalDisplay1[6], PIN\_M24

KamalDisplay1[5], PIN\_Y22

KamalDisplay1[4], PIN\_W21

KamalDisplay1[3], PIN\_W22

KamalDisplay1[2], PIN\_W25

KamalDisplay1[1], PIN\_U23

KamalDisplay1[0], PIN\_U24

KamalDisplay2[6], PIN\_AA25

KamalDisplay2[5], PIN\_AA26

KamalDisplay2[4], PIN\_Y25

KamalDisplay2[3], PIN\_W26

KamalDisplay2[2], PIN\_Y26

KamalDisplay2[1], PIN\_W27

KamalDisplay2[0], PIN\_W28

KamalDisplay3[6], PIN\_V21

KamalDisplay3[5], PIN\_U21

KamalDisplay3[4], PIN\_AB20

KamalDisplay3[3], PIN\_AA21

KamalDisplay3[2], PIN\_AD24

KamalDisplay3[1], PIN\_AF23

KamalDisplay3[0], PIN\_Y19