Josh Gayso, Ben Effron, Jeff Humphrey, Thomas Bonatti

**Addressable Registers**

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Register** | **Address** | **Purpose** | **Register** | **Address** | **Purpose** |
| $M | 000 | Main accumulator. Used for arguments and returned values. | $t0 | 100 | General use. |
| $ra | 001 | Return address | $t1 | 101 | General use. Recommended to be used for flags |
| $sp | 010 | Stack pointer | $t2 | 110 | General use. Recommended for use as a second argument or return value. |
| $at | 011 | Pseudo instructions | $s1 | 111 | Safe use |

**Non-Addressable Registers:**

|  |  |
| --- | --- |
| **Register** | **Purpose** |
| PC | Points to current instruction. |

**I-Type instructions (Immediate):**

I-type instructions will use the main accumulator register, and a 12 bit immediate. Most logic and arithmetic commands will use this.

|  |  |
| --- | --- |
| Op Code [15:12] | Immediate [11:0] |

**R-type (move):**

Operations using registers

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op Code [15:12] | rt (register) [11:9] | rs (register) [8:6] | Func code[5:2] | Unused [1:0] |

**IR-Type(Immediate and Register):**

Branches and load word store word

|  |  |  |
| --- | --- | --- |
| Op code [15:12] | Reg [11:9] | Immediate [8:0] |

**JR-Type [Jump Register]:**

Used for jump register command

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Op Code [15:12] | Reg [11:9] | Unused[8:6] | Func code[5:2] | Unused [4:0] |

Op codes:

|  |  |  |  |
| --- | --- | --- | --- |
| **Op Code** | **Operation** | **Op Code** | **Operation** |
| 0000 | Use function code\* | 1000 | li (load immediate) |
| 0001 | addi (add immediate) | 1001 | ori (or immediate) |
| 0010 | beq (brach equal) | 1010 | andi (and immediate) |
| 0011 | bne (branch not equal) | 1011 | nori (nor immediate) |
| 0100 | J (jump) | 1100 | sll (shift left logical) |
| 0101 | lw (load word) | 1101 | srl (shift right logical) |
| 0110 | sw (store word) | 1110 | beqz |
| 0111 | lui (load upper immediate) | 1111 | bneq |

**Function Codes:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Func Code** | **Operation** | **Func Code** | **Operation** |
| 0000 | add | 1000 |  |
| 0001 | and (logic) | 1001 | slt (set less than) |
| 0010 | or (logic) | 1010 | sub (subtract) |
| 0011 | xor (logic) | 1011 |  |
| 0100 | nor (logic) | 1100 |  |
| 0101 |  | 1101 |  |
| 0110 | copy | 1110 |  |
| 0111 | jr | 1111 |  |

**RTL table**

|  |  |  |  |
| --- | --- | --- | --- |
|  | **R-type (ALU)** | **I-type (ALU)** |  |
| **Inst. Fetch** | IR = mem[PC]  PC = PC + 1 | | |
| **Inst. Decode** | AiA = Reg[IR [7:5] ]  BiB = Reg[IR [4:2] ] | AiA = Reg [0]  BiB = SE [IR [11:0] ] |  |
| **Execution** | ALUout = AiA op BiB | |  |
| **Mem / Reg access 1** | Reg [0] = ALUout | |  |

**Instruction Descriptions**

**Check function code** (op = 0000)

If the op code is 0000, the operation to be executed will be determiner by a 4 bit function code [11:8]

**Addi** **– add immediate** (op 0001, func = xxxx)

signature: addi <12 bit imm.>

I-type

Addi is an I-type operation that adds the given immediate to the accumulator register.

**Beq – Branch equal**  (op = 0010, func = xxxx)

signature: beq $r, <9 bit imm.>

IR-type

Beq branches from PC + 1 if the given register $r and the main accumulator are equal. Can branch +/- 256 lines up or down.

**Bne** - **brach not equal**(op = 0011, func = xxxx)   
signature: beq $r, <9 bit imm.>

IR-type

Bne branches if the given register $r and the main accumulator are not equal. Can branch ± 256 lines from PC+1.

**J – jump** (op = 0100, func = xxxx)

Signature: j <12 bit imm.>

I-type

The 4 most significant digits in the PC will be concatenated with a 12 bit immediate and used as the new PC.

**lw – load word** (op = 0101, func = xxxx)

signature: lw $r[ <9 bit imm.> ]

IR-type

Loads a 16 bit chunk of data from memory at the given address incremented by the immediate.

**sw** **– store word** (op = 0110, func = xxxx)

signature: sw $r [ <9 bit imm.> ]

IR-type

Stores the value in the main register in memory at the given address, incremented by the immediate.

**Li – Load Immediate** (op = 1000, func = xxxx)

Signature: li <12 bit imm.>

I-type

Li stores the given immediate value in the main register.

**ori – or immediate** (op = 1001, func = xxxx)

signature: ori <12 bit imm.>

I-type

ori performs a logical or operation on the value of the main register and the immediate, then stores the value in the main register.

**andi – and immediate** (op = 1010, func = xxxx)

signature: andi <12 bit imm.>

I-type

andi performs a logical and operation on the value of the main register and the immediate, then stores the value in the main register.

**nori – nor immediate** (op = 1011, func = xxxx)

signature: nori <12 bit imm.>

I-type

nori performs a logical nor operation on the value of the main register and the immediate, then stores the value in the main register.

**sll – shift left logical** (op = 1100, func = xxxx)

IR-type

signature: sll $r, <9 bit imm.>

**srl – shift right logical** (op = 1101, func = xxxx)

IR-type

signature: srl $r, <9 bit imm.>

**add – add** (op = 0000, func = 0000)

signature: add $r1, $r2

R-type

Add adds $r1 and $r2 together and stores in the main register. To accumulate, use add $m, $r1.

**and – and** (op = 0000, func = 0001)

signature: and $r1, $r2

R-type

Preforms a logical and operation on two registers, then stores the result in the main register.

**or – or** (op = 0000, func = 0010)

signature: or $r1, $r2

R-type

Preforms a logical or operation on two registers, then stores the result in the main register.

**xor – xor** (op = 0000, func = 0011)

signature: xor $r1, $r2

R-type

Preforms a logical xor operation on two registers, then stores the result in the main register.

**nor – nor** (op = 0000, func = 0100)

signature: nor $r1, $r2

R-type

Preforms a logical nor operation on two registers, then stores the result in the main register.

**copy – copy** (op = 0000, func = 0110)

signature: copy $r1, $r2

R-type

Writes the value of $r1 into register $r2.

**jr – jump register** (op = 0000, func = 0111)

signature: jr $r

JR-type

jumps to the address in the given register.

**slt – set less than register** (op = 0000, func = 1001)

signature: sltr $r1, $r2

R-type

If the main register is less than register $r1, a flag is set to 1 in register $r2. If the main register is equal to or greater than register $r1, a flag is set to 0 in register $r2.

**sub – subtract** (op = 0000, func = 1010)

signature: sub $r1, $r2

R-type

Does operation $m = r1 – r2.

**beqz – branch if zero** (op = 1110)

Signature: beqz <9 bit imm.>

IR-type

If the main register equals zero, branches. This branch command can branch ±256 instructions from PC+1

**bnez– branch if zero** (op = 1111)

Signature: bnez <9 bit imm.>

IR-type

If the main register does not zero, branches. This branch command can branch ±256 instructions from PC+1

**Pseudo Instructions**

**Jal – Jump and Link**

Jump and link jumps to an immediate after backing up the return address and setting a new return address in $ra.

**Assembly Instructions:**

Copy $m, $at # store argument in $at

Copy $ra, $m # load return address

Sw $sp[0] # store return address in memory

Copy $sp, $m # load stack pointer

Addi 1 # update stack pointer

Copy $m, $sp # store new stack pointer

Li [current PC + 4] # load the address of the (current line + 4) to set new $ra, (done in assembler)

Copy $m, $ra # sets $ra to the new return address

Copy $at, $m # restores the argument to the main reg from $at.

J [given immediate] # jumps to desired destination.

**Machine Code:**

0000 000 011 0110 xx

0000 001 000 0110 xx

0110 010 000000000

0000 010 000 0110 xx

0001 000000000001

0000 000 010 0110 xx

1000 [current address, calculated in assembler]

0000 000 001 0110 xx

0000 011 000 0110 xx

0100 [given immediate]

**Swap – swap registers**

Swaps the values between registers

**Assembly Instructions**:

Copy $r1, $at

Copy $r1, $r2

Copy $at, $r1

**Machine Code:**

0000 [r1] 011 0110 xx

0000 [r1] [r2] 0110 xx

0000 011 [r1] 0110 xx

**Lui – Load upper immediate**

Loads an immediate larger than would normally fit in the load immediate command.

**Assembly instructions**:

Li [given immediate]

Sll $m, 4

**Machine code:**

1000 [given immediate]

1100 000 000000100

**Move**

Moves a value from one register to another AND zeros out the source register

**Assembly instructions:**

if r1 = 000:

copy $m, $r2

li 0

else:

copy $r1, $r2

copy $m, $at

li 0

copy $m $r1

copy $at, $m

**Machine code:**

If $r1 = $m (000):

0000 000 [r2] 0110 xx

1000 000000000000

Else:

0000 [r1] [r2] 0110 xx

0000 000 011 0110 xx

1000 000000000000

0000 000 [r1] 0110 xx

0000 011 000 0110 xx

**slti – set less than immediate**

slti sets a 1 in the given register if the value in the main reg is less than the given immediate number.

Slti $r, <12 bit immediate>

**Assembly Instructions**:

copy $m, $r # store $m in given reg. The reg is used to store a flag later, and is therefore safe to overwrite in the pseudoinstruction,

li <immediate> # load immediate

copy $m, $at # place immediate in $at

copy $r, $m # restore $m

slt $at, $r # sets less than.

**Machine Code:**

0000 000 011 <$r> xx

1000 <immediate>

0000 000 011 0110 xx

0000 <$r> 000 0110 xx

0000 011 <$r> 1001 xx

**Datapath Components:**

**PC:** PC is the pointer counter register. It outputs the current instruction address, then stores the next instruction address. It takes in and outputs bits [15:0]. PC also has a PC write signal which prevents the PC to be incremented during cycles when it should not be incremented.

**Adder1**: This adder takes in the current instruction address and adds 1 to it. Both input and output are 16 bits.

**Adder2:** This adder is used for branches and adds the sign extended immediate value from the instruction with the PC. However, this happens AFTER PC has been incremented for this cycle, so the jump will really be from PC of current instruction + branch value + 1

**Instruction Memory:** Instruction memory is a block of memory which contains the instructions in the program. It takes in the current instruction address and outputs the instruction. Takes in and outputs bits [15:0]

**IR:** IR is the instruction register. It takes the instruction output from the Instruction Memory, holds it through the rest of the cycle, and then on the next cycle outputs the instruction that is was fed in. It takes in and outputs bits [15:0]

**Register File:** The bits from the instruction are split up and sent to the appropriate areas of the Register File. The Register File takes in the addresses of the relevant registers from the instruction and outputs the contents of those registers. It also takes in either the resulting value from an operation or an immediate value. The result of all computations are stored in main, so there is no destination register input. The register file outputs the contents stored in the specified registers. The register file has one control signals: reg. Regw is the signal for register writing. If it is turned on the value output by the main ALU is stored in the main register. If it is off writing to the main register is disabled. The ra1 takes in bits [7:5] which chooses the register in both R and IR type instructions. Ra2 takes in bits [4:2], which are chooses the second register for R type instructions. The register file outputs two sets of 16 bits, the contents of the registers.

**ALUinA, ALUinB:** These are both registers that take the output from the register file to store the values between cycles and then output those values to the ALU. Takes in and outputs 16 bits.

**ALU Main:** ALU Main is the main ALU in the processor. The values stored in ALUinA, ALUinB, the sign extended immediate value are inputs for the ALU, and are chosen depending on the type of instruction. The ALU outputs the result of the computation. The ALU takes in ALU control, which tells the ALU which operation to perform. ALU main takes in two sets of 16 bits and outputs a 16 bit answer. It also outputs a 1 bit, isZero which is used for branches.

**ALUOut:** ALUOut is the register that stores the resulting calculations from the ALU. ALUOut stores these results between cycles and outputs the results at the beginning of the next cycle. It takes in the 16 bit results and outputs a 16 bit result.

**Data Memory:** Data memory is the main memory in the computer. Data memory takes in the result from the ALU either retrieves the value stored at that location in memory or stores the value in ALUinA (the value of the main register) at the location in memory in the specified register, and outputs that value. Data memory has a mem write control signal, which allows writing to memory on the correct instructions and turning off writing to memory on instructions that are not supposed to write to memory. Data memory takes in a 16 bit number and outputs the 16 bit result from the read.

**MemOut:** A register that holds the 16 bit output of reads from memory through cycles, and outputs the value stored in it.

**Various Components:** A variety of muxs and sign extends are in use.

Register Transfer Language

\*Ai = ALUin

\*IR-Type is split up between lw/sw, jump, branch, and copy

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | R-Type | I-Type | Load Imm | Jump | Branch | Copy | Load word | Store  Word | Jump | JR | Beqz bnez |
| **Inst Fetch** | IR = Mem[PC]  PC = PC + 1 | | | | | | | | | | |
| **Inst Decode**  **Reg fetch** | AiA = Reg[IR[11:9]]  AiB =  Reg[IR[8:6]] | AiA = Reg[$m]  AiB = SE[IR[11:0]] | Reg[0] =  SEimm[15:0] | AiA = Reg[0]  AiB =  Reg[IR[11:9]] | AiA = Reg[0]  AiB = Reg[IR[11:9]] | AiA = Reg[IR[11:9]] | AiA = Reg[IR[11:9]]  AiB = SE[IR[8:0]] | | PC = PC[15:12] joined with  IR[11:0] | PC = Reg[IR[11:9]] | AiA = Reg[0]  AiB = 0 |
| **Execution** | ALUOut = AiA op AiB | | **LI DONE** | **JUMP DONE** | If (AiA == AiB) (or not equal to)  Then: PC = PC + IR[8:0]  ELSE: PC = PC | Reg[IR[8:6] = AiA | ALUOut = AiA + AiB | | **JUMP DONE** | **JUMP REG DONE** | If (AiA == AiB)  (or not equal to)  Then: PC = PC + IR[8:0]  ELSE: PC = PC |
| **Reg/mem Access** | Reg[0] = ALUOut | |  |  | **Branch Done** | **Copy Done** | MemOut = Mem[ALUOut] | Mem[ALUOut] = Reg[$m] |  |  | **Beqz/bnez done** |
| **Reg/mem Access** |  |  |  |  |  |  | Reg[0] = MemOut | **SW DONE** |  |  |  |
| **lw done** |  |  |  |  |  |  | **lw done** |  |  |  |  |

RTL test plan

This documentation walks through brief examples of how each command in the RTL will work.

**R-Type (ALU Commands)**:

**Initialize:** $m = 10 (0b1010), $t0 = 5 (0b101), PC = 0

**Instructions**: Add $m, $t0 (0000 0000 000 100 XX)

And $m, $t0 (0000 0001 000 100 XX)

Add $m, $t0

**Fetch:** IR = 0000 0000 000 100 XX

PC = 0 + 1

**Decode:** AiA = 10

AiB = 5

**Execute:** ALUout = 10 + 5 = 15

**Mem/Reg:** Reg[0] = 15

And $m, $t0

**Fetch:** IR = 0000 0001 000 100 XX

PC = 1 + 1

**Decode**: AiA = 15 (0b1111)

AiB = 5 (0b0101)

**Execute**: ALUout = 1111 && 0101 = 0101

**Mem/Reg**: Reg[0] = 5

**I-type (ALU commands)**:

**Initialize**: $m = 0, PC = 10

**Instructions**: Addi 15

Ori 32

Addi 15

**Fetch**: IR = 0001 000000001111

PC = 10 + 1 = 11

**Decode**: AiA = 0b0000000000000000

AiB = 0b0000000000001111

**Execute**: ALUout = 15 + 0 = 15

**Mem/Reg**: reg [0] = 15

Ori 32

**Fetch**: IR = 1001 000000100000

PC = 11 + 1 = 12

**Decode**: AiA = 0b0000000000001111

AiB = 0b0000000000100000

**Execute:** ALUout = 15 || 32 = 0000000000101111 = 47

**Mem/Reg:** reg [0] = 47

**Load Immediate**

**Initialize**: $m = 0, PC = 10

**Instructions**: Li 10

Li12

**Fetch**: IR = 1000 000000001010

PC = 10 + 1 = 11

**Decode**: Reg[0] = 10

**Fetch**: IR = 1000 000000001100

PC = 11 + 1 = 12

**Decode**: Reg[0] = 12

**Jump**

**Initialize**: PC = 0x014a

**Instructions**: j 10

J 100

j 10

**Fetch**: IR = 0100 000000001010

PC = 0x014a + 1 = 0x014b

**Decode**: PC = 0x0 concatenate 0x00a = 0x000a

j 100

**Fetch**: IR = 1000 000000001100

PC = 0x000a + 1 = 0x000b

**Decode**: PC = 0x0 concatenate 0x064 = 0x0064

**Branch**

**Initialize**: $m = 0, $t0 = 0, $t1 = 10 PC = 10

**Instructions**: beq $t0, 15

beq $t1, 32

beq $t0, 15

**Fetch**: IR = 0010 000000001111

PC = 10 + 1 = 11

**Decode**: AiA = 0b0000000000000000

AiB = 0b0000000000000000

**Execute**: ALUout = 0 - 0 = 0

PC = 11 + 15 = 26

beq $t1, 32

**Fetch**: IR = 0010 000000100000

PC = 26 + 1 = 27

**Decode**: AiA = 0b0000000000000000

AiB = 0b0000000000001010

**Execute:** ALUout = 0 – 10 = -10

PC = 27

**PC testing**: The PC register must be able to handle jumps, branches, and just regular incrementing. To test the incrementing we could send control signals 0 to both branch and jump, so that PC would

increment normally by one. We could set the clock to some rate, and ‘and’ the PC write control signal

with the third bit of the current PC so that it would increment only once every 4 cycles. If PC successfully

incremented by one on only the fourth cycle we know that both PC is incrementing correctly and that it

only increments when PC write is on. To test jumps we will set the jump mux to on and the branch mux

to off, and we can set the bits on the wires to different test values, including all zeros and all ones, with

the PC wires and see if it the resulting value written to PC is equivalent to the desired jump address.

Finally, we need to test both if turning the branch on will branch to the correct value and if the

comparison comes back false, that even if branch is turned on, the branch value will NOT be added to

the PC. We can do this by first testing branching on and testing the integrity of branching. The ALUOut

input wire will be set to different values, both some positive and negative values, and the zero, one, and

negative one fringe cases. Both the branch and isZero control signals will be turned on, to enable

branching so that the accuracy of the branches can be detected. After the branch passes these tests, we

will turn branch on and isZero off and vice versa to make sure that PC does NOT jump. Additionally, in

this case we need to make sure that PC does not increment a second time when the branch fails.

**Memory Testing:** Memory needs to be tested to make sure that reading and writing from memory can only take place when it is intended. Additionally, we need to make sure that memory will read the right address at the right time. We can test this by setting memory read to on, then putting different values on the IR and the Main register input wires, and change the signal between reading from PC to select instruction memory and reading data values data memory. We also need to see if no value is read from memory when memory read is turned off in both of the aforementioned cases. Secondly, we need to write different values to different locations in memory, including the values -1 and 0, and then on the

next cycle read the value located at the address just written to. If the value we wrote is equal to the

value we read from that location on the next instruction we know that memory is working.

**ALU Control:** To test ALU control we need to make sure that all of the signals from ALU control perform the correct operation. Because our processor has no form of overflow control, we do not care about overflow fringe case, we only need to test on different values that we know will not overflow the ALU. We will test multiple different values on each different operation, including some duplicate and flipped values (i.e. -1 and 1), and make sure that the ALU outputs the expected result. We also need to make sure that isZero has the correct output on every one of these tests, to make sure that branches will work correctly.

**Main Control:** Because ALU control is a state machine, we need to test every single base control signal

type and make sure that the correct control signals are switched to each cycle. We need to make sure

that all R type instructions go through the exact same states, and the same with all of the I type

instructions. We also need to test each of the paths for the IR type instructions, as many of the IR type

instructions have their own unique paths. We need to make sure that every single state path returns to

the base instruction fetch state after completion. Finally, we need to make sure that our processor

catches bad instructions. If an instruction does not match any of the designated paths, it sets all control

signals to 0 until a valid instruction is read from memory.

Because we are using Xilinx’s built in muxes and registers, the supplied register file, and an already

written and tested ALU we do not need to test these components individually.

**Component Integration Testing**

**PC and Memory Integration Testing:** PC needs to be able to successfully read a value out of memory and output it to IR register. We will do this by sending an always on bit both into PCWrite and

MemRead, and setting PC to a base value of a pre-determined memory address, with a value at it that

we already know, with subsequent known values, and let it run. If IR successfully receives and stores the

instruction read from memory by PC, then we will know that PC and memory work together.

**IR, Reg File, ALU Control and ALU testing:** The register file needs to be able to successfully output data to AiA and AiB, which is then correctly processed by the ALU, which is given the correct operation at the correct time. We will test this by directly setting the values of the registers in the register file, and then making sure that the data from IR gets decoded correctly, by reading data from the right registers and outputting that data to AiA and AiB. Then in the next cycle the ALU gets the operation instruction and performs the correct operation, sending the expected output to ALUOut. This will test mostly the ability for the register file and ALU to work together correctly, and to make sure the timing on the ALU control is also correct. We also need to flip the ior bit on the mux going into AiB and do the same thing as described above, except making sure that the read value for register AiB does not get written to AiB, and instead the correct immediate value extracted from IR gets written to AiB instead. After that the same ALU operations and control will be tested, making sure that the datapath has the correct timing and ability to deal with immediate values.

**Memory and Register File:** We need to make sure that the instruction read from memory can be

correctly decoded when going to the register file. We can do this by feed different memory locations

directly into memory, with known values at those locations, and make sure that value gets written to IR.

On the next cycle, we need to read the values going into the read in 1 and read in 2 ports on the register

file to confirm that they are reading from the correct registers. The muxes going into the register file

also need to be examined, switching the mux going into r1in to make sure that the register file will

correctly ignore the address bits going into it and will read from the main register located at position 0.

Secondly, the mux for the wires going into r2in needs to be tested on both values to make sure that the

register file reads from the correct part of the instruction bits from IR. We also need to make sure that

memory can correctly write to the register file. The register write address is not handled by memory, so

we can manually switch that to different registers, and we can input the memory location into memory

to get out a known value. We need to try to write to all of the different registers, to make sure that

memory does not only write correctly to a single register and not work on others.

**ALUOut, Memory, and Register File:** This test will check to make sure that both memory and the ALU can write back a value to the Register file. Because the only instructions that don’t write back to main are jr, swap, and copy, none of which involve either data memory nor the ALU, and because we know that our register file works correctly from previous testing, we can always just check the contents of the main register at the end of each cycle to make sure that the correct value was saved in main. Because we know that the ALU works correctly, we can simply set ALUOut to hold different values and check at the end of each cycle if the Main register is now storing that value. Reading from memory and then writing to the register file takes two cycles, so we will have to set up an improvised control for this test. This control will be a very simple state machine, setting regWrite to 0 if it was previously a 1 and back to a 1 if it was previously a zero. The reverse would be true for memory read. In this way writing to main is turned off when reading from data memory, and writing to main is turned on on the next cycle. The reverse would be true for main memory. We can inject a memory address with a known value into

memory, and then after the next cycle see if main is storing that value.

**Control and ALU Control:** While previous test will have shown that these two components work

individually, it is critically important that they work together. No output data needs to be read, but

different instructions will be sent to the control. The control states will be checked every cycle, and the ALU Control signals will also be checked, making sure that they are off when no operation should be

going on and that they are turned on on the corrected cycle when the register values are supposed to be

going through the ALU. We will make sure that the execution cycle for all of our state diagram paths

match up with sending the correct ALU signal.

**Datapath Implementation Testing**

Because all of the components will be individually tests, and all of the components will be integration

tested as described above, the most important tests will be making sure that everything in the datapath

runs together in sync. When the different component groups are wired together as shown in the

datapath diagram we will run test instructions, and make sure that all sections can complete their job

and store their data in a register before the next cycle begins. Different instructions of different lengths

will be run to make sure the instructions terminate at the correct cycle and reset so a new instruction

can be read in. We will check the states of all wires during each cycle, making sure that the control stays

in sync with the operations happening in the datapath, and cycle times will be adjusted to the lowest

time that ensures that all operations will be completed. We will send multiple different instructions of

each type through the processor and check the resulting locations of the data in the datapath to make

sure that not only were any branches, jumps, or arithmetic operations were completed correctly, but

also that the values were stored in the correct locations before control resets and a new instruction is

read in. We will also have to look over every wire and compare them with our datapath diagram to

ensure that no wire is hooked up incorrectly. This would mostly be checked after the aforementioned

tests are performed, if a problem is detected.

**Implementation details**

**Control:** Our processor has two control units, a main control unit that is the source of many one or two

bit multiplexer control signals, and an ALU controller that is the source of a 4 bit ALU op code. Both

controllers take a 4 bit op code and a 4 bit function code from the instruction, however, the function code

is ignored unless the op code equals 0.

The main control unit is a state machine implemented in behavioral Verilog. When the state machine’s

current state is Fetch (state 0), the controller will read the op code to determine its next state. If the op

code is 0, then the controller will select the next state based on the func code. If the control unit is not in

the “fetch” state, it’s next state is based on its current state and the op and func codes are ignored. This

means the controller will continue to execute the given instruction, whether the op code changes or not.

The ALU control unit is much simpler. It simply takes in an op and function code, ignoring the func code

unless the op code is zero, ALU uses the same arithmetic fuctions for different commands, the ALU

controller acts as a interpreter for the op codes fed into the ALU. The ALU control is implemented in

behavioral Verilog.

**Arithmetic Logic Unit**: Our ALU is implemented in behavioral Verilog. The ALU takes a 4-bit control signal,

two 16-bit arithmetic inputs (A and B), one 16-bit arithmetic output, and one single-bit “isZero” output

used in comparisons. The ALU can perform 12 operations, as follows: Add, or, xor, and, nor, sll (shift left

logical), srl (shift right logical), sub (subtraction), nand, mult (multiplication), eq0 (set if equals 0), slt (set

if less than). The ALU is used in any arithmetic or logic based instructions.

**Register File**: Our Register File was built in behavioral Verilog. It contains eight 16-bit registers. Two

registers can be read at once, however only one address can be written to at once. In addition to the 2

read out signals, there is a main register out signal that always outputs the data from register [0], the

accumulator. This data is wired straight to dataIn on our memory, since the only way to store data on the

stack is directly from this register using the store word command.

**Adder**: This adder is implemented in behavioral Verilog. The adder is used at one points in the datapath,

to add 1 to PC every instruction to simplify the datapath by saving wires and muxes going into the ALU.

**Sign Extender:** The sign extender is written in behavioral Verilog, and is used to sign extend 9 and 12 bit

immediates into 16 bit immediates that can be used in the ALU.

**Memory**: We used the IP(Core Generator & Architecture Wizard) to generate a block memory for our

project, exactly as described in Lab 7. This memory will be double clocked, to prevent any delays in data

access. There will be one memory, which will store instructions and a stack for use by programs. The

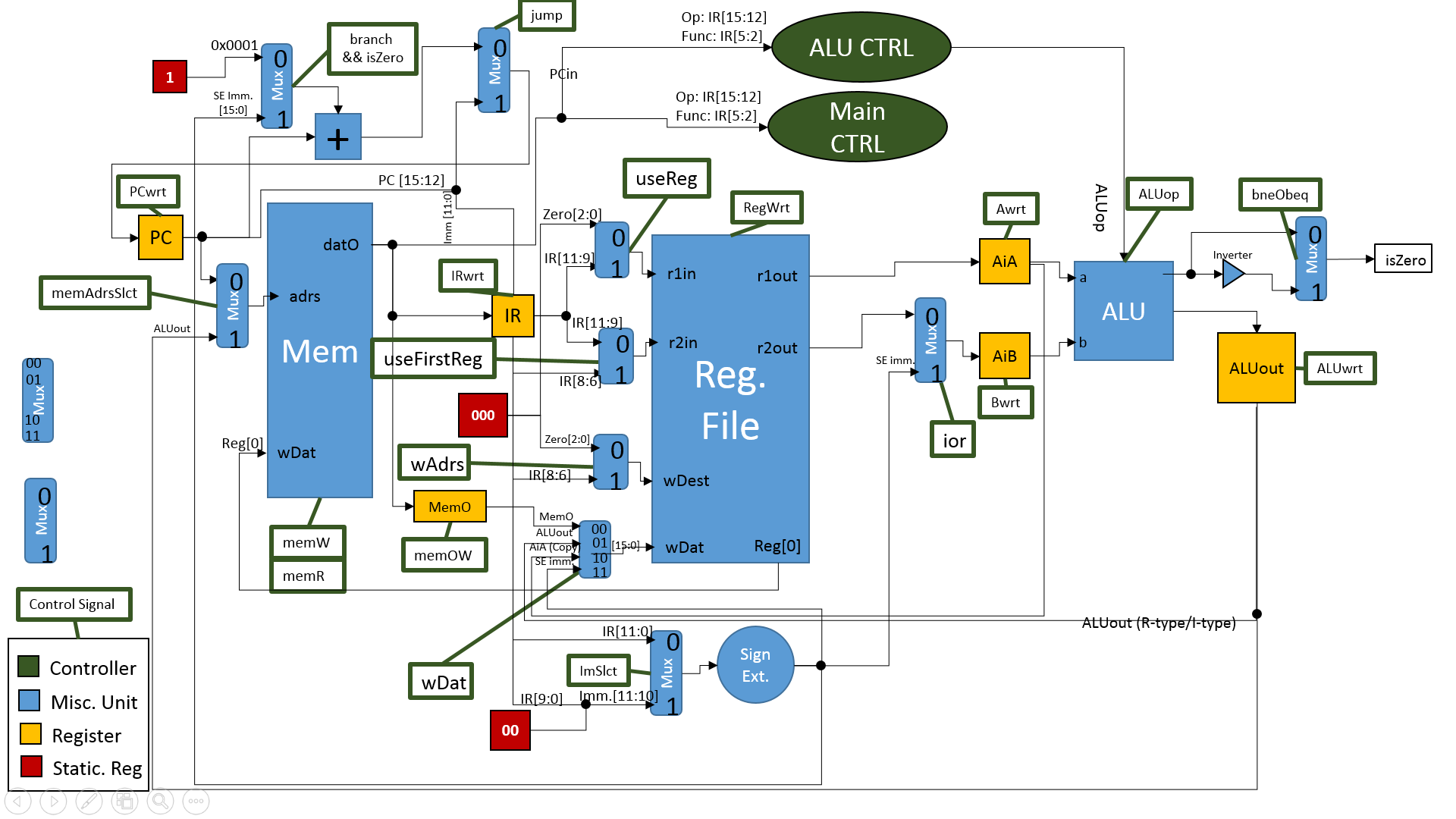
memory has an access depth of 16 and an access width of 1024.

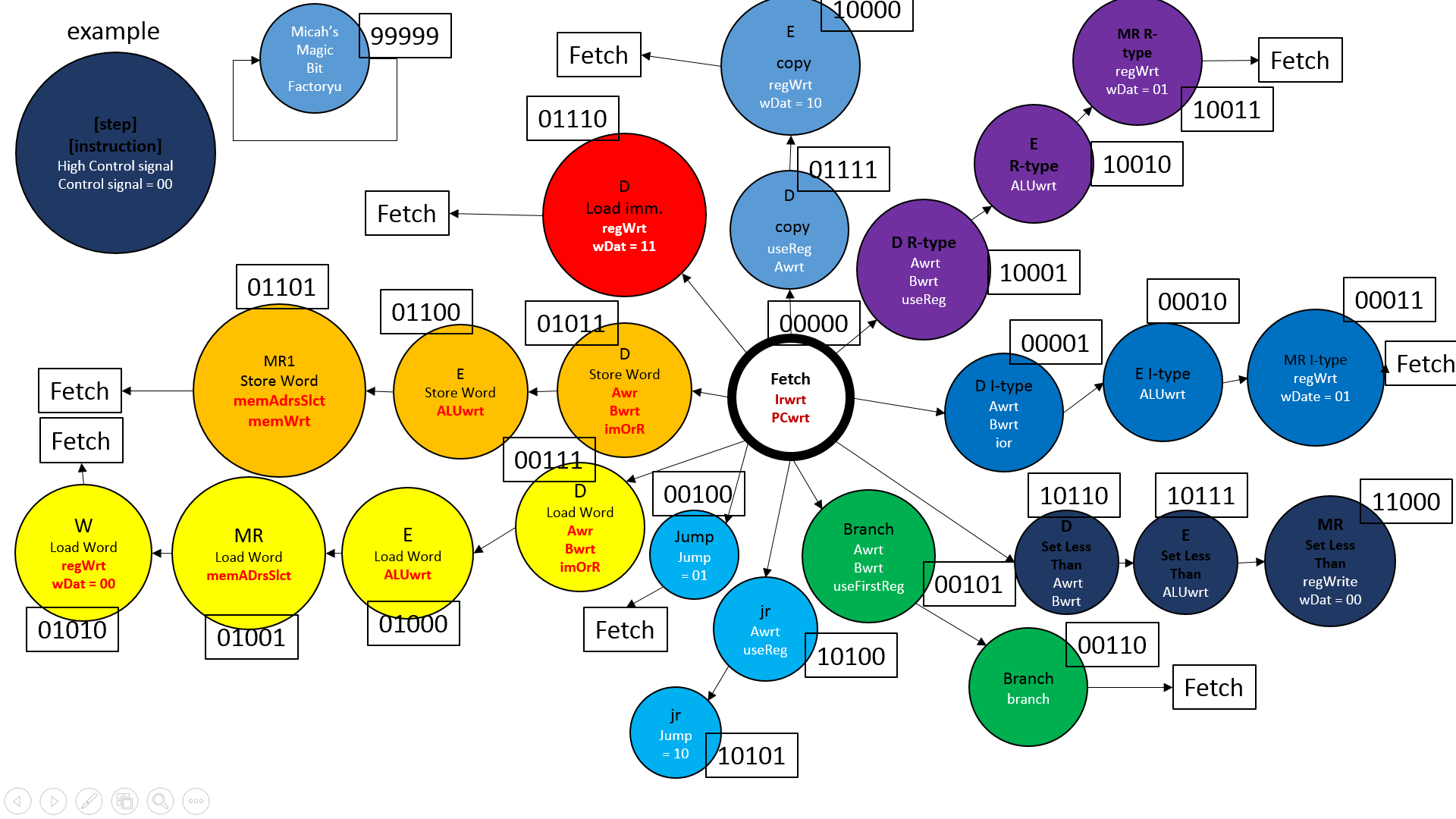
**Registers and Multiplexers**: Registers and multiplexers are built into the syntax of behavioral level Verilog,

and we have not implemented our own versions.

**Datapath:** We have not implemented a datapath design in Xilinx yet, however we plan to use behavioral

level Verilog code to link the modules we have designed and tested.







**Tests for Control Units**

For our control unit, we must test extensively every possible input for the OP code, along with the two unique FUNC codes, with all of the possible combinations of RESET and CLK bits.

We must test OP codes ‘b0000 through ‘b1111 along with FUNC codes ‘b0111 and ‘b0110 and assure that the correct combination of control signals are output.

We must also confirm that when other FUNC codes are put into the system, and the proper R-type OP code is input, that the R type control signals are output.

We must make sure that everything happens on the posedge of the CLK.

We also must make sure that when RESET is high, that all of the outputs are 0.

