Rose-Hulman Institute of Technology

SHER-VI

Multi-Cycle Memory-Memory

CSSE-232

Nicolas Kireyev, Mateo Olson, Ethan Wilson

Computer Architecture I | CSSE 232

Stephen Sher

4/3/2024

The memory to memory (memory-memory) architecture that this CPU design adopts takes priority in memory accessing through executing operations between memory locations. This CPU design, however, does use allocated slots in memory and a single register for built in purposes, such as a return address (S0), a dedicated zero (S1), a space for arguments and return values (S2-S5), and stack pointer as the single register. Functionally, these offsets are similar to registers in RISC-V, but with relative addresses to SP rather than directly referencing registers. These memory locations start at the bottom of the stack 0x00000000 for S0 and increment by 4 for each following space. The overall CPU design follows a 32-bit architecture, including instructions and items in memory. In addition, it utilizes multi-cycle processing for faster instruction speeds.

The formation for instructions consists of four format types: Set (S) for storing a large immediate within an offset of SP or for jumping by an offset while storing the return address in the SP offset, Huge (H) for handling functions by adding an offset to SP, Element (E) for basic arithmetic with reg, and Reach (R) for branch comparisons.

Measuring performance will be determined by the program’s execution time; including memory read and write speed, clock speed, and the number of instructions. For simulation we used Quartus to design the modules and ModelSim to run tests. GitHub was used for version control.

|  |  |
| --- | --- |
| **Relative Offset:** | **Purpose:** |
| ra | Return Address |
| zero | Dedicated Zero |
| a0-a3 | Arguments/Return Values |
| s0+ | General Purpose Protected Memory Space |

|  |  |
| --- | --- |
| **Type:** | **Description:** |
|  | **Bit Allocation:** |
| Set  00 | Set is used to set an offset of SP as a signed immediate value |
| (S) | [31destOffset23|22immediate2|1opcode0] |
| Huge  01 | Huge type adds the offset to SP and stores it in TSP (choice bit = 0), or subtracts the offset from SP and stores it in SP (choice bit = 1). Addsp is used before function calls to set the new SP for that function once the next branch is called. Subsp is used after returned from a function to put SP back. |
| (H) | [31large offset3|2choice bit2|1opcode0] |
| Element  10 | Element type is used for operations involving three offsets of SP, with one destination offset and two source offsets. For example, these could be arithmetic operations such as add or shift left |
| (E) | [31destOffset23|22offset14|13offset5|4funct32|1opcode0] |
| Reach  11 | Reach type is used for operations involving two offsets of SP and an offset of SP that stores a memory address (offset 0 is used for return address), used for conditional branching/jumping with comparisons of two offset values of SP |
| (R) | [31destOffset23|22offset14|13offset5|4funct32|1opcode0] |

**English description of instructions:**

|  |  |
| --- | --- |
| **Instruction** | **Description:** |
| **Set** |  |
| make | Loads the large immediate value at the offset of SP |
|  |  |
| **Huge** |  |
| addsp | Adds the shifted unsigned immediate value to TSP |
| subsp | Subtracts the shifted unsigned immediate value to SP |
|  |  |
| **Element** |  |
| add | Adds the values in both SP offsets (sr1 and sr2) and stores the result at the destination SP offset (dest). |
| sub | Subtracts the value at the second SP offsets (sr2) from the first SP offset (sr1) and stores the result at the destination SP offset (dest). |
| xor | XORs the values at both SP offsets (sr1 and sr2) and stores the result at the destination SP offset (dest). |
| or | ORs the values at both SP offsets (sr1 and sr2) and stores the result at the destination SP offset (dest). |
| and | AND’s the values at both SP offsets (sr1 and sr2) and stores the result at the destination SP offset (dest). |
|  |  |
| sll | Shifts the value at the first SP offset (sr1) by the value at the second SP offset (sr2) and stores the result at the destination SP offset (dest). |
| srl | Shift the value at the first SP offset (sr1) by the value at the second SP offset (sr2) and stores the result at the SP offset (dest); shifts zeroes in from the left. |
| sra | Shift the value at the SP offset (sr1) by the value at the second SP offset (sr2) and stores the result at the destination SP offset (dest); shifts the sign bit in from the left. |
|  |  |
| **Reach** |  |
| beq | Adds TSP into SP. Stores the value of PC + 4 into SP offset 0. Compares the values in both SP offsets (sr1 and sr2) and branches to the SP offset (mem addr) if they are equal. |
| bne | Adds TSP into SP. Stores the value of PC + 4 into SP offset 0. Compares the values in both SP offsets (sr1 and sr2) and branches to the SP offset (mem addr) if they are not equal. |
| blt | Adds TSP into SP. Stores the value of PC + 4 into SP offset 0. Compares the values in both SP offsets (sr1 and sr2) and branches to the SP offset (mem addr) if sr1 is less than sr2 |
| bge | Adds TSP into SP. Stores the value of PC + 4 into SP offset 0. Compares the values in both SP offsets (sr1 and sr2) and branches to the SP offset (mem addr) if sr1 is greater than sr2 |
| jeq | Compares the values in both SP offsets (sr1 and sr2) and branches to the SP offset (mem addr) if they are equal. |
| jne | Compares the values in both SP offsets (sr1 and sr2) and branches to the SP offset (mem addr) if they are not equal. |
| jlt | Compares the values in both SP offsets (sr1 and sr2) and branches to the SP offset (mem addr) if sr1 is less than sr2 |
| jge | Compares the values in both SP offsets (sr1 and sr2) and branches to the SP offset (mem addr) if sr1 is greater than sr2 |

**Practical description of instructions:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Type** | **Opcode** | **Instr** | **Symbolic Description:** | **Syntax:** |
| S | 00 | make | Offset[dest] = imm | make dest, imm |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Opcode** | **Choice Bit** | **Instr** | **Symbolic Description:** | **Syntax:** |
| H | 01 | 0 | addsp | SP += 2\*imm | addsp imm |
| H | 01 | 1 | subsp | SP -= 2\*imm | subsp imm |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Funct3** | **Opcode** | **Instr** | **Symbolic Description:** | **Syntax:** |
| E | 000 | 10 | add | Offset[dest] = Offset[sr1] + Offset[sr2] | add dest, sr1, sr2 |
| E | 001 | 10 | sub | Offset[dest] = Offset[sr1] - Offset[sr2] | sub dest, sr1, sr2 |
| E | 010 | 10 | xor | Offset[dest] = Offset[sr1] ^ Offset[sr2] | xor dest, sr1, sr2 |
| E | 011 | 10 | or | Offset[dest] = Offset[sr1] | Offset[sr2] | or dest, sr1, sr2 |
| E | 100 | 10 | and | Offset[dest] = Offset[sr1] & Offset[sr2] | and dest, sr1, sr2 |
|  |  |  |  |  |  |
| E | 101 | 10 | sll | Offset[dest] = Offset[sr1] << Offset[sr2] | sll dest, sr1, sr2 |
| E | 110 | 10 | srl | Offset[dest] = Offset[sr1] >> Offset[sr2] | srl dest, sr1, sr2 |
| E | 111 | 10 | sra | Offset[dest] = Offset[sr1] >> Offset[sr2] se | sra dest, sr1, sr2 |

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Type** | **Funct3** | **Opcode** | **Instr** | **Symbolic Description:** | **Syntax:** |
| R | 000 | 11 | beq | 0[SP] = PC+4  if(Offset[sr1]== Offset[sr2]):PC=Offset[addr] | beq addr, sr1, sr2 |
| R | 001 | 11 | bne | 0[SP] = PC+4  if(Offset[sr1]!= Offset[sr2]):PC=Offset[addr] | bne addr, sr1, sr2 |
| R | 010 | 11 | blt | 0[SP] = PC+4  if(Offset[sr1]< Offset[sr2]):PC=Offset[addr] | blt addr, sr1, sr2 |
| R | 011 | 11 | bge | 0[SP] = PC+4  if(Offset[sr1]> Offset[sr2]):PC=Offset[addr] | bge addr, sr1, sr2 |
|  |  |  |  |  |  |
| R | 100 | 11 | jeq | if(Offset[sr1]== Offset[sr2]):PC=Offset[addr] | jeq addr, sr1, sr2 |
| R | 101 | 11 | jne | if(Offset[sr1]!= Offset[sr2]):PC=Offset[addr] | jne addr, sr1, sr2 |
| R | 110 | 11 | jlt | if(Offset[sr1]< Offset[sr2]):PC=Offset[addr] | jlt addr, sr1, sr2 |
| R | 111 | 11 | jge | if(Offset[sr1]> Offset[sr2]):PC=Offset[addr] | jge addr, sr1, sr2 |

**Multicycle RTL description of instructions:**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Instr.** | **Cycle 1** | **Cycle 2** | **Cycle 3** | **Cycle 4** |
| **Set** |  |  |  |  |
| make | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | Mem[IRT[15:7]<<1 + SP] <= IRT[6:0] | IRO[15:2]SE |  |  |
|  |  |  |  |  |
| **Huge** |  |  |  |  |
| addsp | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | TSP<=IRT[15:0] | IRO[15:3]<<1 + SP |  |  |
| subsp | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | SP<=SP-IRT[15:0] | IRO[15:3]<<1 |  |  |
|  |  |  |  |  |
| **Element** |  |  |  |  |
| add | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | ALUOUT<=A+B | Mem[IRT[15:7]<<1 + SP <= ALUOUT |
| sub | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | ALUOUT<=A-B | Mem[IRT[15:7]<<1 + SP <= ALUOUT |
| xor | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | ALUOUT<=A^B | Mem[IRT[15:7]<<1 + SP <= ALUOUT |
| or | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | ALUOUT<=A|B | Mem[IRT[15:7]<<1 + SP <= ALUOUT |
| and | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | ALUOUT<=A&B | Mem[IRT[15:7]<<1 + SP <= ALUOUT |
| sll | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | ALUOUT<=A<<B | Mem[IRT[15:7]<<1 + SP <= ALUOUT |
| srl | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | ALUOUT<=A>>B | Mem[IRT[15:7]<<1 + SP <= ALUOUT |
| sra | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | ALUOUT<=A>>B  (sign extends) | Mem[IRT[15:7]<<1 + SP <= ALUOUT |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
|  |  |  |  |  |
| **Reach** |  |  |  |  |
| beq | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | CMP <= A==B? 1:0 z <= TSP + 2 | Mem[TSP] <= PC Mem[z] <= 0  SP <= TSP  PC <= C |
| bne | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | CMP <= A!=B? 1:0 z <= TSP + 2 | Mem[TSP] <= PC Mem[z] <= 0  SP <= TSP  PC <= C |
| blt | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | CMP <= A< B? 1:0 z <= TSP + 2 | Mem[TSP] <= PC Mem[z] <= 0  SP <= TSP  PC <= C |
| bge | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | CMP <= A>=B? 1:0 z <= TSP + 2 | Mem[TSP] <= PC Mem[z] <= 0  SP <= TSP  PC <= C |
| jeq | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP] C<=Mem[IRT[15:7]<<1 + SP] | CMP <= A==B? 1:0 | PC <= C |
| jne | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | CMP <= A!=B? 1:0 | PC <= C |
| jlt | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | CMP <= A< B? 1:0 | PC <= C |
| jge | IRO<=Mem[PC] IRT<=Mem[PC+2] PC<=PC+4 | A<=Mem[IRT[6:0] | IRO[15:14]<<1 + SP]  B<=Mem[IRO[13:5]<<1 + SP]  C<=Mem[IRT[15:7]<<1 + SP] | CMP <= A>=B? 1:0 | PC <= C |

**Translating to Machine Code:**

For all types the 2 bit opcode becomes the right 2 bits. For E and R types the next 3 bits to the left are the funct3 and for the H type the next bit is the choice bit. For S and H types the immediate comes next and for E and R types the source 1 offset and source 2 offset go next respectively. E, R, and S types put the destination offset (word offset not byte offset) next.

**Addressing Modes:**

All immediates are together so no arranging has to be done because the other arguments are already lined up nicely. Branching gets a global address from a register and sets PC to it.

**Calling Conventions:**

Before calling a function arguments are placed in S(N+3) to S(N+6) assuming S(N) was the latest offset used. The caller is responsible for adding to SP to compensate for all offsets it uses. The return address is set to the offset S0 when a branch instruction is used. The offset S1 is automatically set to a zero. Upon returning, it is the callee’s responsibility to manage placing the return values in S1-S3, and the caller’s responsibility to subtract the same offset that was added to SP beforehand.

**Relprime:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Assembly** | **Machine Code** | **Comments** |
| 0x0000 | make a1, 2; | 00000001100000000000000000001000 | //Set m (SP3) to 2 |
| 0x0004 | make a2, 1; | 00000010000000000000000000000100 | //Temp 1 at SP4 |
| 0x0008 | make a3 WHILE; | 00000010100000000000000001010000 | //Set up address of WHILE for jump |
| 0x000C | make s0 PRIMERETURN; | 00000011000000000000000011010000 | //Set up address of PRIMERETURN for jump |
| 0x0010 | make s1, GCD; | 00000011100000000000000011110000 | //Set up address of GCD for jump |
| 0x0014 | WHILE: add s4, a0, zero; | 00000000100000001000000101000010 | //Set SP10 (future a0) to n, labelled |
| 0x0018 | add s5, a1, zero; | 00000000100000001100000101100010 | //Set SP11 (future a1) to m |
| 0x001C | addsp 8; | 00000000000000000000000001000001 | //add 7 offsets of space to TSP |
| 0x0020 | beq s1, a1, a1; | 00000001100000001100000011100011 | //Go to GCD, store return address in 0 |
| 0x0024 | subsp 8; | 00000000000000000000000001000101 | //Remove the 7 offsets after return from GCD |
| 0x0028 | jeq s0, s4, a2; | 00000010000000101000000011010011 | //if SP9 (return value a0) = 1, skip to return |
| 0x002C | add a1, a1, a2; | 00000010000000001100000001100010 | //add 1 to m |
| 0x0030 | jeq a3, zero, zero; | 00000000100000000100000010110011 | //unconditional repeat of while loop |
| 0x0034 | PRIMERETURN: add a0, a1, zero; | 00000000100000001100000001000010 | //set S2 to m, labelled |
| 0x0038 | jeq ra, zero, zero; | 00000000100000000100000000010011 | //Return to RA |
| 0x003C | GCD: make a2, GCDWHILE; | 00000010000000000000000101000000 | //Set up GCDWHILE for while loop, labelled |
| 0x0040 | make a3, GCDGE; | 00000010100000000000000110000000 | //Set up GCDGE for checking b ≥ a |
| 0x0044 | jne a2, a0, zero; | 00000000100000001000000010010111 | //jump to GCDWHILE if a != 0 |
| 0x0048 | add a0, a1, zero; | 00000000100000001100000001000010 | //Set S4 to b |
| 0x004C | jeq ra, zero, zero; | 00000000100000000100000000010011 | //Return to RA |
| 0x0050 | GCDWHILE: jeq ra, a1, zero; | 00000000100000001100000000010011 | //Return if b = 0 (a is already the default return value), labelled |
| 0x0054 | jge a3, a1, a0; | 00000001000000001100000010111111 | //Skip to GCDGE if b ≥ a |
| 0x0058 | sub a0, a0, a1; | 00000001100000001000000001000110 | //a = a - b |
| 0x005C | jeq a2, zero, zero; | 00000000100000000100000010010011 | //unconditional repeat of while loop |
| 0x0060 | GCDGE: sub a1, a1, a0; | 00000001000000001100000001100110 | //b = b - a |
| 0x0064 | jeq a2, zero, zero; | 00000000100000000100000010010011 | //return to RA |

**Assembly Language Fragments:**

|  |  |  |  |
| --- | --- | --- | --- |
| **Address** | **Assembly** | **Machine Code** | **Comments** |
| 0x0000 | make a1, 158; | 00000001100000000000001001111000 | //Assigning an immediate value to an offset |
|  |  |  |  |
| 0x0000 | add a1, a2, a3 | 00000010100000010000000001100010 | //Basic add instruction |
|  |  |  |  |
| 0x0000 | make a1, 1 | 00000001100000000000000000000100 | //Iteration |
| 0x0004 | make a2, 0 | 00000010000000000000000000000000 |  |
| 0x0008 | make a3, FINISH | 00000010100000000000000001110000 | //setting up addresses for loops |
| 0x000C | make a4, WHILE | 00000011000000000000000001000000 |  |
| 0x0010 | WHILE: jeq a3, a1, a0 | 00000001000000001100000010110011 | //loops value of S2 times |
| 0x0014 | add a1, a1, a2 | 00000010000000001100000001100010 | //increment counter |
| 0x001C | jeq a4, a2, a2 | 00000010000000010000000011010011 |  |
| 0x0020 | FINISH: |  |  |
|  |  |  |  |

0x00FF (SP)

0xFFFF

Instructions

Stack

Memory

0x0000 (PC)

16 Bit Words

2 Word Long

Instructions

ra

zero

a0-a3

s0, s1, ...

1 Word Long

Memory Spaces

0x0101

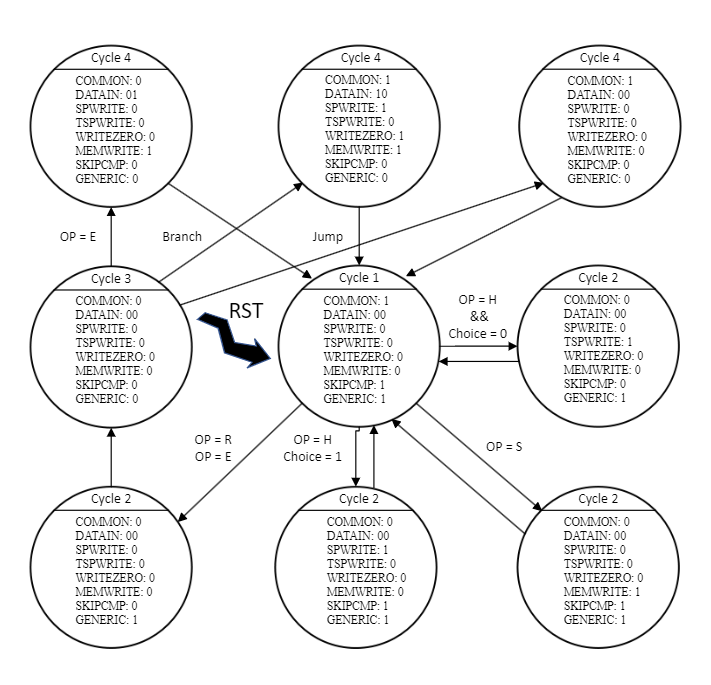
0x0103

0x010B

Instruction

0x0004

|  |  |
| --- | --- |
| Control Signal Name | Description |
| COMMON | Enables writing to IRO and IRT and enables writing to PC when cmp or SKIPCMP is 1. Also is the select bit for the destination address when writing to Mem and the select bit for the input of SP. |
| SKIPCMP | Enables writing to Mem, PC, or SP regardless of the output of the ALU comparison (not branching). Still requires that their respective enable bits are 1. |
| MEMWRITE | Enables writing to Mem when cmp or SKIPCMP is 1. |
| WRITEZERO | Enables writing 0 to Mem[TSP + 2] when cmp is 1. |
| TSPWRITE | Enables writing to TSP. |
| SPWRITE | Enables writing to SP when cmp or SKIPCMP is 1. |
| DATAIN[1:0] | Select bit to determine what data should be written to Mem[destination address]. |
| GENERIC | Select bit for the PC input, the adder that adds 2 to the variable input, and enables C to be written to. |



|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Components | Inputs | Outputs | Behavior | RTL Symbols |
| Memory |  | IRO[15:0]  IRT[15:0]  Arg1[15:0]  Arg2[15:0]  Arg3[15:0] | Holds all 16 bit words. Has a dedicated space for instructions. And a dedicated space for data. Accessing data is SP relative and SP will change whenever a function is called | Mem |
| ALU | A[15:0]  B[15:0]  Funct3[2:0] | ALUOUT[15:0]  cmp[0:0] | Arithmetic logic unit used process an operation based on the two inputs and the desired control | ALU |
| Adder | Input0[15:0]  Input1[15:0]  Sub[0:0] | Sum[15:0] | Performs the addition of two inputs, and returns their sum as an output. Can be switched to subtraction with the sub bit. | + |
| Registers | Input[15:0]  Overwrite[0:0]  CLK[0:0] | Output[15:0] | Updates stored value only when enabled, outputs value on every cycle. | TSP  Sp  PC  A  B  C  IRO  IRT  ALUOUT  cmp |
| Muxes | Input0[15:0]  Input1[15:0]  \*Input2[15:0]  Select[0:0] | Output[15:0] | Takes in multiple values and outputs one value depending on the selector bit. |  |
| Control | IRO[15:0]  IRT[15:0]  CLK[0:0]  Reset[0:0] | COMMON[0:0]  DATAIN[1:0]  SPWRITE[0:0]  TSPWRITE[0:0]  WRITEZERO[0:0]  MEMWRITE[0:0]  SKIPCMP[0:0]  GENERIC[0:0] | Takes in the instruction and outputs control bits. |  |

See Appendix B for Datapath

Appendix A

Assembler Code (Java, Eclipse IDE)

package assembler;

import java.io.File;

import java.io.FileNotFoundException;

import java.util.Scanner;

import java.math.BigInteger;

import java.util.ArrayList;

import java.util.HashMap;

/\*

Converts S.H.E.R VI assembly language from the Assembly.txt file into machine code. The program will stop running when the file is done or when the end keyword is reached. Comments can be put in the assembly file with \\. Extra spaces in the program will be removed by the assembler. Nothing should separate arguments besides spaces. A bad instruction type will be caught, and skipped over after an error message is displayed. If a good instruction type has bad arguments, the program will either: catch the error and skip over it after displaying the error, or if the error follows instruction format (like make g5 40), the program will run it and interpret it as a default instruction type (will treat all characters besides 'a' as 's', g5 == s5 when determining the memory location). Comments that start with \* instead of \\ will be displayed to the console

\*/

public class AssemblerNew {

String[] opCodes = new String[4];

Scanner sc;

File file;

// The total binary instruction

String instr;

// The current binary instruction to be concatenated

String tempInstr;

// The full assembly instruction

String assembly;

// The operation to be run add, beq, etc.

String operation;

// Creates the assembler

public static void main(String[] args) {

AssemblerNew assembler = new AssemblerNew();

}

// Starts the Program

public AssemblerNew() {

initOpCodes();

initScanner();

run();

}

// Creates list of Opcodes

public void initOpCodes() {

this.opCodes[0] = "00";

this.opCodes[1] = "01";

this.opCodes[2] = "10";

this.opCodes[3] = "11";

}

// Creates the Scanner

public void initScanner() {

this.file = new File("Assembly.txt");

try {

this.sc = new Scanner(this.file);

} catch (FileNotFoundException e) {

// TODO Auto-generated catch block

e.printStackTrace();

}

}

// Given the assembly string, sets instr as the OpCode

public void findOp() {

if (assembly.substring(0, 5).equals("addsp") || assembly.substring(0, 5).equals("subsp")) {

instr = "01";

} else if (assembly.substring(0, 4).equals("make")) {

instr = "00";

} else if (assembly.charAt(0) == 'b' || assembly.charAt(0) == 'j') {

instr = "11";

} else {

instr = "10";

}

}

// Reduces the number of spaces in between arguments to 1

public String reduceSpaces(String assembly) {

boolean sawSpace = false;

String newAssembly = "";

int lastIndex = 0;

for (int i = 0; i < assembly.length(); i++) {

if (assembly.charAt(i) == ' ') {

if (sawSpace) {

newAssembly += assembly.substring(lastIndex, i);

lastIndex = i + 1;

}

sawSpace = true;

} else {

sawSpace = false;

}

}

newAssembly += assembly.substring(lastIndex);

return newAssembly;

}

// Loops through all of the code and calls functions

public void run() {

// Grabs the first line

assembly = sc.nextLine();

assembly = assembly.trim();

// Runs until end is reached or the file ends

while (!assembly.equals("end")) {

boolean isText = false;

// If assembly is long enough, check for comments to be displayed

if (assembly.length() > 0) {

if (assembly.charAt(0) == '\*') {

isText = true;

}

}

if (!isText) {

// Check for undisplayed comments

if (assembly.length() >= 2) {

if (assembly.substring(0, 2).equals("//")) {

// Code here got removed, the space is reserved in case something

// is to be added when an undisplayed comment is called in the future.

} else if (assembly.length() >= 5) {

assembly = reduceSpaces(assembly);

findOp();

findFunct3();

// Check if the OpCode is wrong

// By default instructions are set to operational instructions (opCodes[2])

// If a different instruction type is not found

// If no other funct3 is found the default operation is "jge"

// the jge operation is for branch instructions, not operational instructions

// this conflict causes the line the be skipped, (a bad instruction type)

if (!(instr.equals(opCodes[2]) && operation.equals("jge"))) {

addInstr();

// Once instructions are checks, arguments are ran, if there is any error

// Such as trying to parse a string that is not an int as an int

// An error will be displayed and the line will be skipped

try {

if (operation.equals("make")) {

make();

} else if (operation.equals("addsp") || operation.equals("subsp")) {

sp();

} else {

threeOffsets();

}

System.out.println(this.instr);

} catch (Exception e) {

System.out.println("Incorrect Argument Format: " + assembly);

}

} else {

System.out.println("Could not find instruction, going to next line: " + assembly);

}

} else {

System.out.println("Instruction too short, going to next line: " + assembly);

}

} else {

System.out.println("Instruction too short, going to next line: " + assembly);

}

} else {

// Prints the line for displayed comments

System.out.println(assembly);

}

// gets the next line if possible, if not sets assembly to "end" to end the

// while loop

if (sc.hasNextLine()) {

assembly = sc.nextLine();

assembly = assembly.trim();

} else {

assembly = "end";

}

}

sc.close();

}

// runs the code for E and R types (three offsets)

public void threeOffsets() {

int isOr = operation.equals("or") ? 1 : 0;

String imm;

int numZeros = 0;

int isZero = 0;

for (int i = 0; i < 3; i++) {

isZero = 0;

if (assembly.charAt(4 - isOr + 3 \* i - 3 \* numZeros + 5 \* numZeros) == 'z') {

isZero = 1;

}

imm = assembly.substring(4 - isOr + 3 \* i - 3 \* numZeros + 5 \* numZeros,

4 - isOr + 3 \* i - 3 \* numZeros + 5 \* numZeros + 2 - isZero \* 2 + 4 \* isZero);

if (isZero == 1) {

numZeros++;

}

tempInstr = convertImm(Integer.toString(regNum(imm)), 9);

addInstr();

}

}

// runs the code for the sp instruction

public void sp() {

String imm;

imm = assembly.substring(6);

this.tempInstr = convertImm(imm, 29);

addInstr();

}

// takes in the "register" name and returns the offset int

public int regNum(String reg) {

if (reg.equals("ra")) {

return 0;

} else if (reg.equals("zero")) {

return 1;

} else if (reg.charAt(0) == 'a') {

return 2 + Integer.parseInt(reg.substring(1, 2));

} else {

return 6 + Integer.parseInt(reg.substring(1, 2));

}

}

// runs the code for the make instruction

public void make() {

String imm;

int immStart;

int offSet;

if (assembly.charAt(5) == 'z') {

immStart = 10;

offSet = regNum("zero");

} else {

immStart = 8;

offSet = regNum(assembly.substring(5, 7));

}

imm = assembly.substring(immStart);

this.tempInstr = convertImm(imm, 21);

addInstr();

this.tempInstr = convertImm(Integer.toString(offSet), 9);

addInstr();

}

// Given that tempInstr is set to the next part, updates instr

public void addInstr() {

this.tempInstr += this.instr;

this.instr = this.tempInstr;

}

// Converts an the string of an immediate (base 10) into a binary string given the length it should be after sign extending

public String convertImm(String imm, int signExtend) {

String tempImm;

char lastBit = imm.charAt(0);

if (lastBit == '-') {

lastBit = '1';

imm = imm.substring(1);

} else {

lastBit = '0';

}

imm = Integer.toBinaryString(Integer.parseInt(imm));

if (lastBit == '1') {

for (int i = 0; i < imm.length(); i++) {

if (imm.charAt(i) == '1') {

imm = imm.substring(0, i) + '0' + imm.substring(i + 1);

} else {

imm = imm.substring(0, i) + '1' + imm.substring(i + 1);

}

}

int immlen = imm.length();

imm = Integer.toBinaryString(Integer.parseInt(imm, 2) + 1);

if (imm.length() > immlen) {

imm = imm.substring(immlen - imm.length());

} else if (imm.length() < immlen) {

String zeros = "";

for (int i = 0; i < immlen - imm.length(); i++) {

zeros += '0';

}

zeros += imm;

imm = zeros;

}

}

// sign extension

int immlen = imm.length();

for (int i = 0; i < signExtend - immlen; i++) {

tempImm = "";

tempImm += lastBit;

tempImm += imm;

imm = tempImm;

}

return imm;

}

// sets the tempInstr to the funct3 or choice bit (if there is one) and sets the operation as the operation name

public void findFunct3() {

if (instr.equals(opCodes[0])) {

tempInstr = "";

this.operation = "make";

return;

} else if (instr.equals(opCodes[1])) {

if (assembly.charAt(0) == 'a') {

tempInstr = "0";

this.operation = "addsp";

} else {

tempInstr = "1";

this.operation = "subsp";

}

return;

} else {

if (assembly.substring(0, 2).equals("or")) {

this.operation = "or";

tempInstr = "011";

} else {

this.operation = assembly.substring(0, 3);

if (this.operation.equals("add")) {

this.operation = "add";

this.tempInstr = "000";

} else if (this.operation.equals("sub")) {

this.operation = "sub";

this.tempInstr = "001";

} else if (this.operation.equals("xor")) {

this.operation = "xor";

this.tempInstr = "010";

} else if (this.operation.equals("and")) {

this.operation = "and";

this.tempInstr = "100";

} else if (this.operation.equals("sll")) {

this.operation = "sll";

this.tempInstr = "101";

} else if (this.operation.equals("srl")) {

this.operation = "srl";

this.tempInstr = "110";

} else if (this.operation.equals("sra")) {

this.operation = "sra";

this.tempInstr = "111";

} else if (this.operation.equals("beq")) {

this.operation = "beq";

this.tempInstr = "000";

} else if (this.operation.equals("bne")) {

this.operation = "bne";

this.tempInstr = "001";

} else if (this.operation.equals("blt")) {

this.operation = "blt";

this.tempInstr = "010";

} else if (this.operation.equals("bge")) {

this.operation = "bge";

this.tempInstr = "011";

} else if (this.operation.equals("jeq")) {

this.operation = "jeq";

this.tempInstr = "100";

} else if (this.operation.equals("jne")) {

this.operation = "jne";

this.tempInstr = "101";

} else if (this.operation.equals("jlt")) {

this.operation = "jlt";

this.tempInstr = "110";

} else {

this.operation = "jge";

this.tempInstr = "111";

}

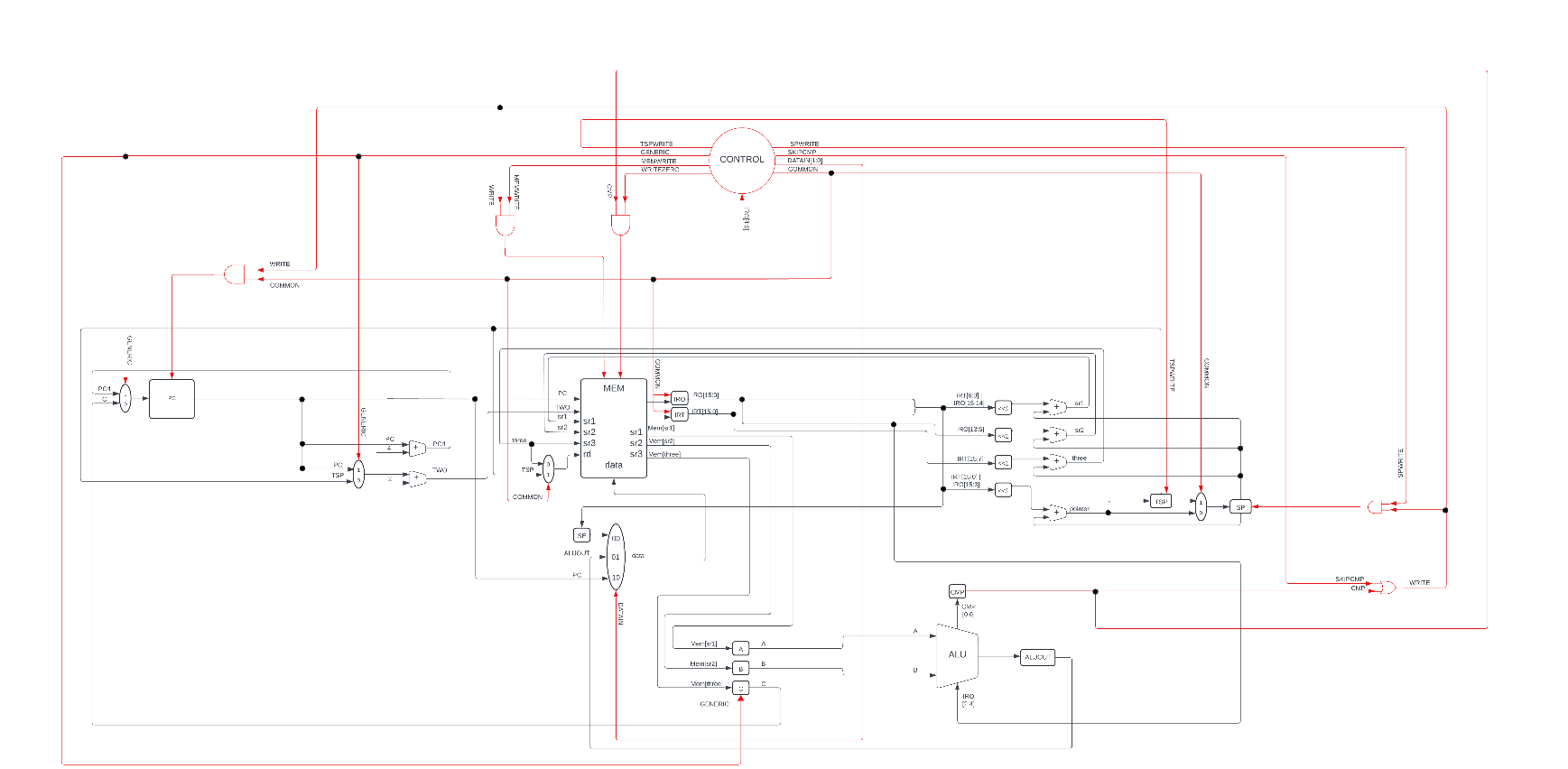
}

}

}

}

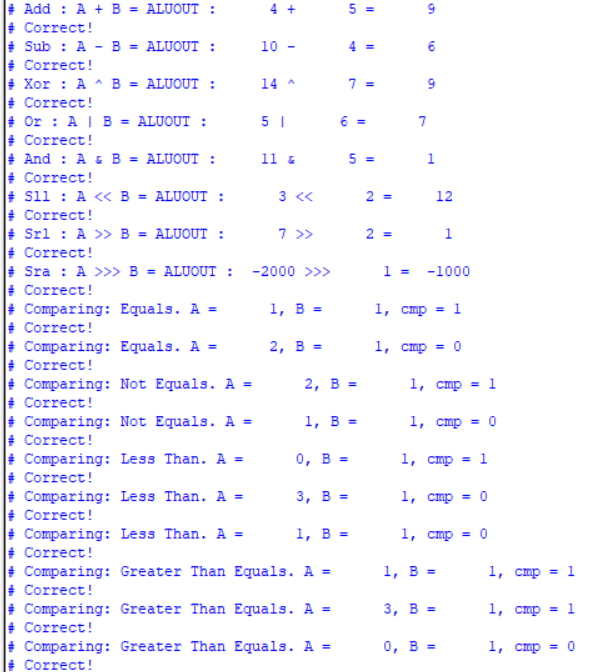
Appendix B

Datapath

Appendix C

Component Testing (Quartus, ModelSim, Verilog)

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Components | Files | Test Description | Testers | Date of Test Completion | Proof of Test Completion |
| ALU | ALU.qws  ALU.v  ALU.v.bak  ALU\_inst.v  ALU\_TEST.v  ALU\_TEST.v.bak  ALU\_inst.v | The ALU component tests verify the arithmetic and logical operations. The testbench ALU\_TEST.v is used to insert different inputs to ensure the ALU outputs expected results. The operations tested include Add, Sub. Xor, Or, And, Sll, Srl, and Sra. | Wilson | 05/03/2024 | Figure 1 |
| CONTROL | CONTROL\_TEST.v  CONTROL\_TEST.v.bak  Control Test.cr.mti  Control Test.mpf  SHER\_VI\_CONTROL.qws  SHER\_VI\_CONTROL\_inst.v  vsim.wlf | The CONTROL component tests verify the control signals for the processor. The testbench CONTROL\_TEST.v inserts different control signal inputs to ensure the components achieve expected results. | Wilson | 05/03/2024 | Figure 2 |
| SL | SL.v  SL.v.bak  Sl.qws  test\_SL.v  test\_SL.v.bak | The SL component tests verify the shift left operation for data shifting. The testbench test\_SL.v is used to insert different inputs to ensure the SL outputs expected results. | Olson | 05/01/2024 | Figure 3 |
| ADDER | adder.qpf  adder.v  adder.v.bak  test\_adder.v  test\_adder.v.bak | The ADDER component tests verify its adding operation. The testbench test\_adder.v is used to insert different inputs to ensure the ADDER outputs expected results. | Olson | 05/03/2024 | Figure 4 |
| MEM | MEMORY\_TEST.v  MEMORY\_TEST.v.bak  Memory.v  Memory.v.bak  Memory\_Test.cr.mti  Memory\_Test.mpf  Memory\_inst.v  SHER\_VI\_MEMORY.qws  vsim.wlf | The MEM component tests verify its read and write operations. The testbench MEMORY\_TEST.v is used to insert different inputs to ensure the MEM outputs expected results. | Wilson | 05/09/2024 | Figure 5 |
| MUX | mux.qws  mux1b2b.v  mux1b2b.v.bak  tb\_mux1b2.v  tb\_mux1b2.v.bak | The MUX component tests verify its selection operations. The testbench tb\_mux1b2.v is used to insert different inputs to ensure the MUX outputs expected results. | Olson | 05/01/2024 | Figure 6 |
| REGISTER | register.qws  register.v  register.v.bak  test\_register.v | The REGISTER component tests verify its operations. The testbench test\_register.v is used to insert different inputs to ensure the REGISTER outputs expected results. | Olson | 05/01/2024 | Figure 7 |

Figure 1

A screenshot of a computer code

Description automatically generatedFigure 2

Figure 3

Figure 4

A white paper with blue text

Description automatically generatedFigure 5

Figure 6

Figure 7