Design

Joseph Brown
Jack Hallam
Joshua Laesch
Armando Luja Salmeron

Instruction Format

- We have one type of instruction, a JIPO-Type (jump, immediate, push, offset) instruction
- The first four bits of each instruction are the OP code
- There are 11 bits for an immediate/address
- The Op code tells the processor everything it needs to know
- All instructions use the same instruction type

JIPO

Op code 5 bits	Address/Immediate 11 bits

Translating to Machine Code

- By hand:
  - Look up the op code for the instruction.
  - pushui and pushli use 8 bit immediates so you can get a full 16 bit immediate (when used together)
  - dupo uses the immediate value as the offset below the stack pointer; 1 means 1 below the pointer, etc. So dupo 0 just duplicates the top of the stack.
  - All other operations just use the op code, the rest doesn't matter
- Using our website:
  - http://www.rose-hulman.edu/~lujasaa/CompilerAssembler/
  - The assembler converts all pseudo instructions into instructions that have op codes.
  - For the .coe files in machine code in our project, there are .txt files with the same name that have the assembly translation.

Instruction Descriptions

Note: 1. Instructions are written as "Instruction Argument". If there is no argument, "Argument" is blank. 2. sp is the stack pointer, "sp-0" is the value at the "top" of the stack (the stack grows up). "sp-1" is the value at one address lower than "sp-0". The lowest address is the bottom of our stack. 3. The instructions described in this section have OP codes, pseudo instructions do not. They are translated by the assembler. 4. Op code in ( )		
pop (00000)	no arguments	"deletes" the value on top of the stack by decrementing the stack pointer by 1; the value is not held/stored in another place
pushui (00001)	8bitImm	pushes 8bitImm followed by 8 zeros to the top of the stack, stack pointer increases by 1 ie. 1111111100000000
pushli (00010)	8bitImm	pushes the 8bitImm preceded by 8 zeros to the top of the stack, stack pointer increases by 1 ie. 0000000011111111
push (00011)	no arguments	replaces the current 16 bit address at the top of the stack with the value of that address from data memory, stack pointer is not changed
add (00100)	no arguments	replaces the value on the stack at sp-1 with the sum of the two values at sp-1 and sp-0, then pop once

or (00101)	no arguments	replaces the value on the stack at sp-1 with the “bitwise or” of the two values at sp-1 and sp-0, then pop once
neg (00110)	no arguments	replaces the value on top of the stack with its negative value, (flips the bits and adds one)
dupo (00111)	11bitImm	duplicate the value at sp - 11bitImm and push that value on top of the stack The net movement of the stack pointer is +1.
j (01000)	no arguments	changes the value of PC to the value that is currently on top of the stack, then pops once (PC = addr on top of stack)
beq/bne (01001/01010)	no arguments	compares the values on the stack at sp-1 and sp-2, if the condition is true it sets PC = address at sp-0 (the top of the stack) , otherwise PC = PC +1. Then it pops 3 times (this removes the address and the two values, so sp = sp - 3). The immediate value of this instruction is set to 3 by the assembler.
slt (01011)	no arguments	if the value on the stack at sp - 1 is less than the value at sp - 0, it replaces the value at sp - 1 with a 1, otherwise it replaces that value with a 0. then it pops once (for either case)
and (01100)	no arguments	replaces the value on the stack at sp+1 with the “bitwise and” of the two values at sp-1 and sp-0, then pop once
store (01101)	no arguments	This stores the value at sp-1 into the address at sp-0 in data memory, then pops two times (so sp = sp - 2). The immediate value of this instruction is set to 2 by the assembler.
repo (01110)	11bitImm	This replaces the value at sp - 11bitImm with the current value at the top of the stack. The stack pointer does not move.
wait (01111)	no arguments	Does nothing until input is received. This is accomplished by keeping PC=PC and continue checking every cycle for a flag that alerts that input has been entered. The input replaces what is on top of the stack. The net movement of the stack pointer is 0.
output (10000)	no arguments	Reads the value on the top of the stack and outputs it. The net movement of the stack pointer is 0.

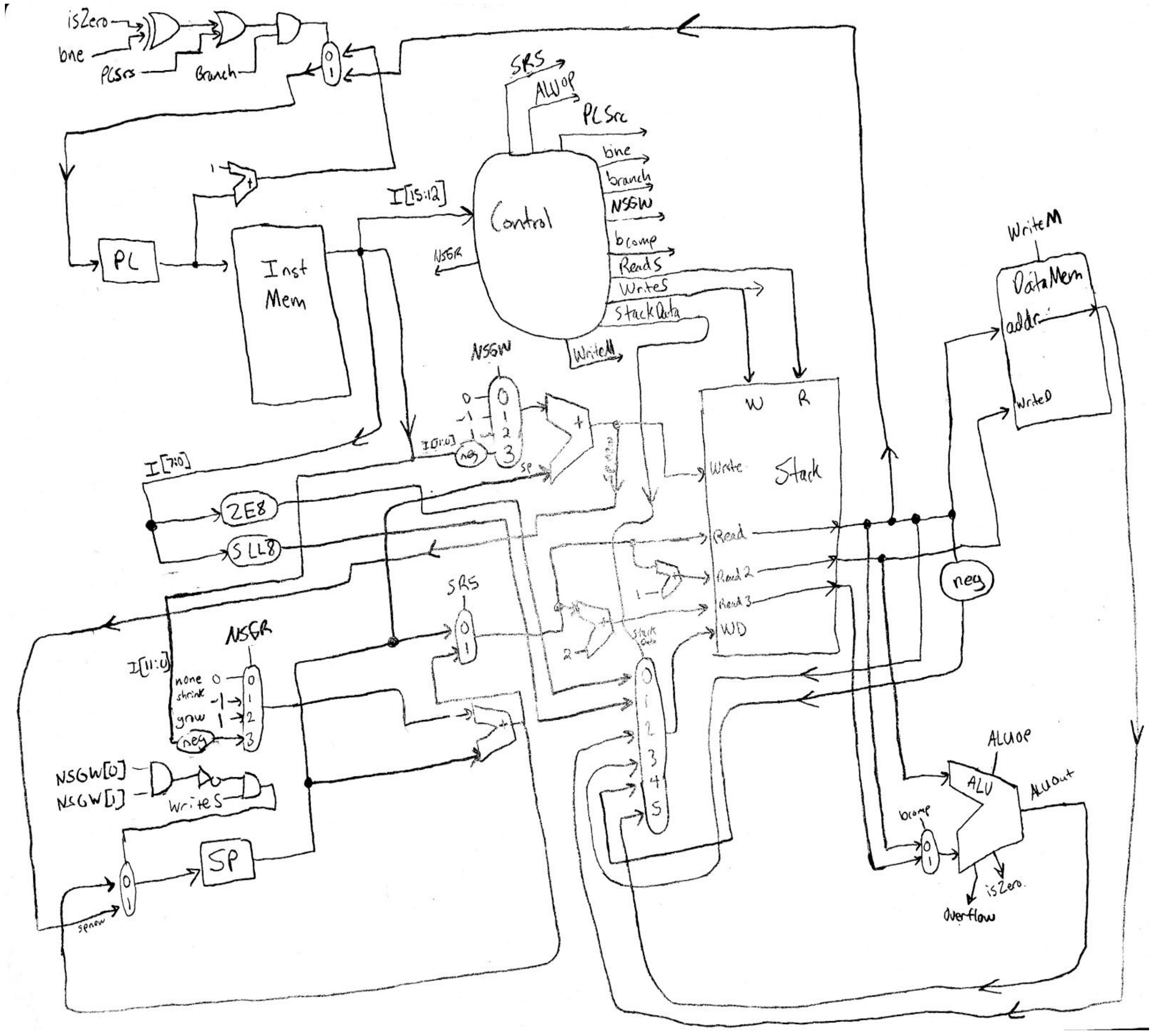
Pseudo-Instruction Descriptions

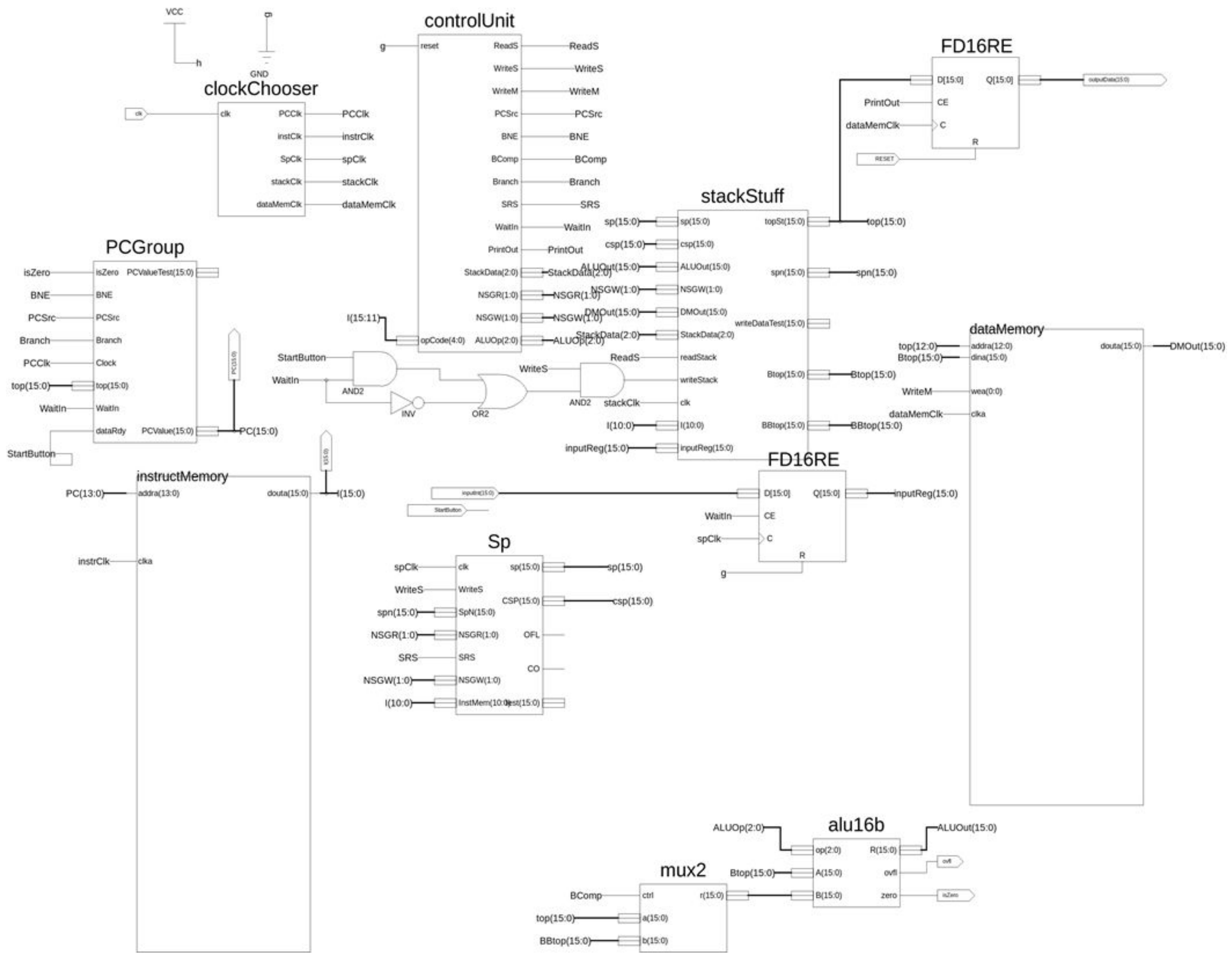
Notes:
1. “val” is a 16 bit immediate or address.
2. “val[15-8]” represents the bits indexed 15 - 8 of “val”.
3. “{op}” is used to represent an operation, such as “add”, “and”, or “or”.
4. “dest” is the 16 bit address in data memory where the result of an operation will be stored, “src” is also a 16 bit address in data memory where a value will be pulled from and used in the operation
5. “(src1)” represents the value at the data memory address “src1”, similarly, “(dest)” represents the value at the data memory address “dest”; src1 does not equal (src1).

Pseudo Instruction:	Translates to:	Description
pushi val	pushui val[15-8] pushli val[7-0] or	This pushes a 16 bit immediate or address on the stack. The net movement of the stack pointer is +1.

push src	pushui src[15-8] pushli src[7-0] or push	This pushes the value at the given address(src) onto the stack. The net movement of the stackpointer is +1.
sub dest, src1, src2	pushi src1 push pushi src2 push neg add pushi dest store	(dest) = (src1) - (src2) The net movement of the stack pointer is 0.
sub src1, src2	pushi src1 push pushi src2 push neg add	The result of “(src1) - (src2)” is pushed on top of the stack. The net movement of the stack pointer is +1.
sub src1	pushi src1 push neg add	This replaces the value on top of the stack with “the value on top of the stack - (src1)”. The net movement of the stack pointer is 0.
add dest, src1, src2 and dest, src1, src2 or dest, src1, src2	pushi src1 push pushi src2 push {op} pushi dest store	(dest) = (src1) {op} (src2) The net movement of the stack pointer is 0.
add src1, src2 and src1, src2 or src1, src2	pushi src1 push pushi src2 push {op}	The result of “(src1) {op} (src2)” is pushed on top of the stack. The net movement of the stack pointer is +1.
add src1 and src1 or src1	pushi src1 push {op}	This replaces the value at the top of the stack with “the value at the top of the stack {op} (src1)”. The net movement of the stack pointer is 0.
addi val1 subi val1 andi val1 ori val1	pushi val1 {op}	*{op} is “add”,, “and”, or “or”. This replaces the value at the top of the stack with “the value at the top of the stack {op} val1”. The next movement of the stack pointer is 0.

Single-Cycle Datapath Sketch





Component Descriptions

- Clock (not shown): It's just a 1 or 0 that changes after a designated time (however long our longest instruction will take).
- Clock-Chooser: Takes the clock, and sends out signals to different components.

IF counter is equal to:	Do:
0	Clock-Up for Instruction Memory and SP
1	Clock-Up for Stack
2	Clock-Up for Data Memory and PC
3	Clock-Down for ALL components

- PC: program counter. A register that hold s the address of the current instruction in data memory that is being executed. It's updated after each instruction is run.
- Instruction Memory: This block of memory is where instructions will be loaded and read from. It is 16 bitswide, and the depth is 15000..
- Control unit: The control unit will output control bit values based on the input, which is a 5 bit opcode from the current instruction.
- SP : This is the stack pointer register that holds the current value of the address that is the top of the stack.
- ALU: This component takes two 16 bit numbers and 3 bits of control. Depending on the control bits, different arithmetic or logic operations are done on the inputs. One 16 bit number is output.
- Adders: These components take two 16 bit numbers and add their values. One 16 bit number is output.

- Stack: This is a register file of 64 registers with a pointer that keeps track of the top of the stack. Each block in this file represents one register. It was created in Verilog using an array of registers.
- Data Memory: This component is slow, but large storage. It would best be suited for storing large amounts of data, possibly arrays.
- Multiplexers: These components take several inputs. They use (a) control bits(s) to select the correct input, and forward it on in a single bus.
- Zero Extender: This component takes 8 bits as input. It sets those bits to be the least significant 8 bits of the output. Eight "0" bits are output in the 8 most significant bits.
- Left Shifter: This component shits bits to the left (less significant bits become more significant). The least significant bits become 0.
- Negator: This component takes 16 bits of input and outputs the two's compliment negation.

Datapath Explained

When the clock strikes, PC will emit the address it is holding and send it into instruction memory to be read. The instruction at that address is shot out at multiple places. The opcode of the instruction goes into control. Control then sends out all of the appropriate control flags to make the instruction work. 8 bits of the immediate are sent into the ZE and SLL and those results are sent into the mux that chooses what data is written to the stack. 11 bits of the immediate are sent to the mux that chooses what is added to sp to read and to the mux that chooses what is added to sp to write. The stack then reads from sp, sp-1, and sp-2. Whether or not all 3 of these values are used depends on the instruction. If data memory is used then the value at sp is used to read or write to that address. The write data for data memory is at sp-1. The values from the stack can also be put into an ALU and depending on the instruction will perform a calculation. The value out of data memory, ALU, and the negator also go into the mux that chooses what is written to the stack. The final step is that the stack will be written to depending on the instruction and sp is changed to point at the top of the stack.

Register transfer level (RTL):

- Single-cycle RTL. For simplicity, we made multiple steps and registers go in this RTL. This makes it look like multi-cycle but our processor really isn't, it is single-cycle. This is to be able to logically think through what each instruction does.

Instruction Execution RTL

Step	pop	push	pushli	pushui

Instruction	IR = InstMem[PC]
	PC = PC + 1			

Stack		A=Stack[sp]		
ALU/Mem		MDR = Mem[A]		

Write/Sp	sp=sp-1	Stack[sp]=MDR	sp = sp +1
			Stack[sp] = ZE(InstMem[7-0])
				sp = sp + 1
				Stack[sp] = (InstMem[7-0] << 8)

Step	dupo	repo	neg

Instruction	IR = InstMem[PC]
	PC = PC + 1		

Stack	A = Stack[sp + InstMem[11-0]]	A = Stack[sp]	A = Stack[sp]

ALU/Mem			

Write/Sp	sp = sp + 1
	Stack[sp] = A
		Stack[sp + InstMem[11-0]]	Stack[sp] = Neg(A)

Step	**j**	**beq**	**bne**

Instruction	IR = InstMem[PC]
	PC = PC + 1		

Stack	A = Stack[sp]	A = Stack[sp]
		B = Stack[sp + 1]
		C = Stack[sp + 2]	

ALU/Mem	PC = A	if(B == C) then
		PC = A
			if(B != C) then
			PC = A

Write/Sp	sp = sp - 1	sp = sp - 3	

Step	**store**	**logical/arithmetic operations**

Instruction	IR = InstMem[PC]
	PC=PC+1	

Stack	A = Stack[sp]
	B = Stack[sp + 1]
		A = Stack[sp]
		B = Stack[sp + 1]

ALU/Mem	DataMem[A] = B	ALUOut = A op B
Write/Sp	sp = sp - 2	Stack[sp + 1] = ALUOut sp = sp - 1

Testing

- Testing the components
  - To test the if the overflow bit is working in the ALU, we would try to add very big numbers (i.e. 60,000 and 60,000) and then have the overflow from the ALU be an output and check to see if it is 1.
  - To test the data memory, we could have a program with a bunch of stores and pushes which will put stuff in memory and then get it back out and see if what we put in equals what we got out.
  - To test if the instruction memory is holding the right instructions, we could add a register after instruction memory and every cycle when and instruction comes out of it, check the register and see if that is the correct instruction.
  - A good way to test if jump worked would be to have a label in our assembly language and call a jump to that label and see if PC was changed.
  - To test our stack, we will need to test all of the instructions and see that the values on the stack change. We can do this by writing to the stack by using push/pushi and then checking that the correct register in the stack is updated. Similarly, for logic and arithmetic functions, we would give 2 values to do the operation on and see if they are put on the stack correctly and then check if the result gets put on the stack. This would be easy to see with a testbench since we could just see what values are being read and written and when. We would test what the stack does when the read and write control bits are on/off and see if it works correctly

- Putting the components in groups
    - We put the smaller components into larger groups so that our overall datapath would be easier to test.
    - We put PC, its logic gates, and an adder in a group. We made sure the control bits worked correctly and PC was updated correctly.
    - SP, the NSGR mux, the adder, and logic gates were in another group. We made sure SP was being updated correctly and that the logic worked correctly.
    - The stack and all the adders, muxes, extenders, negators, etc. were put in a group. We made sure the stack was working with all instructions that used it and that it was being updated correctly.
    - The only things not still in a group are the 2 memories and the ALU stuff. These will be added to the larger datapath. Also, anything still to be created (like I/O and exception handling stuff) will be added to the larger datapath, or if necessary, made as a separate group.
- Testing the datapath (as a whole)
    - The first thing to do to test the datapath is to test pushli, pushui, and pop. When testing pushli and pushui see if the value was put on the stack and when testing pop see if it was taken off. Once that is accomplished test all the other instructions that use the stack such as dupo and repo by seeing if the stack modifies as desired. Then test instructions that use the ALU. For example, add would need to take two values off of the stack and add them together and put the answer on the stack. We would check the stack after and see if the stack has the answer. Finally check store and push and see if data memory is implemented correctly by calling store at multiple locations and see if push will return the same values.
    - We can make a testbench to try and test all the instructions on the whole datapath. If there are errors, then we may have to fix our datapath, or determine if parts of our components are incorrect.t
- Testing the control unit
    - The control unit was implemented using cases in verilog. Each op code was a different case and the control bits were set according to the op code.
    - To test the control unit, we just had to check if the control bits were set properly. We created a test bench that went through every instruction and made sure that all of the controls were set like they were supposed to. The control bits that must be set for each instruction can be seen below in the control flags table.
- Testing the RTL
    - To test the RTL, one would use a combination of the RTL table and the picture of the datapath. First, draw the current state of the stack. Then go through each step in the RTL table and trace it out in the datapath. Look and see if the control bits are set to what they need to be set by looking at the finite state machine. If they are set correctly and if the stack ends what is expected, then it works. For example, for an add instruction we see if the 2 operands are taken off the stack and that their sum is put back
    - on the stack.
    - We would also want to see if the correct output come out of each component with specified inputs. This would test if the control bits are set correctly and if the component itself works correctly. For example, for pushui and pushli, we could see if the correct 16 bit immediate is put on the stack (see if the shifter and zero extender work). Another test would be with store, we see if the address and value get taken off the stack and see if the value gets put at that address in data memory. Similar tests can be done for each instruction and component to see if our logic was correct when we implemented them.

Control Flags for Each Instruction

Note: The number in parentheses tells the number of bits

	ReadS	WriteS	Stack Data (3)	NSGR (nothing shrink grow read) (2)	NSGW (...write) (2)	WriteM	PC Src	BNE	BComp	Branch	ALUOp (3)	SRS (Stack read source)	WaitIn	PrintOut
pop	0	0	X	01	X	0	0	X	X	0	X	X	0	0
push	1	1	010	00	00	0	0	X	X	0	X	X	0	0
pushli	0	1	000	X	10	0	0	X	X	0	X	X	0	0
pushui	0	1	001	X	10	0	0	X	X	0	X	X	0	0
dupo	1	1	011	11	10	0	0	X	X	0	X	1	0	0
neg	1	1	100	00	00	0	0	X	X	0	X	X	0	0
store	1	0	X	11	X	1	0	X	X	0	X	0	0	0
j	1	0	X	01	X	0	1	X	X	1	X	0	0	0
beq	1	0	X	11	X	0	0	0	1	1	110	0	0	0
bne	1	0	X	11	X	0	0	1	1	1	110	0	0	0
or	1	1	101	00	01	0	0	X	0	0	001	X	0	0
and	1	1	101	00	01	0	0	X	0	0	000	X	0	0
add	1	1	101	00	01	0	0	X	0	0	010	X	0	0
slt	1	1	101	00	01	0	0	X	0	0	111	X	0	0
repo	1	1	011	00	11	0	0	X	X	0	X	X	0	0
wait	0	1	110	X	00	0	0	X	X	0	X	X	1	0
output	1	0	X	00	X	0	0	X	X	0	X	X	0	1

Description of Control Flags:

- ReadS: This will be set to 1 when we are reading from the stack
- WriteS: This will be set to 1 when we are writing to the stack
- Stack Data: Controls what is written to the stack
- NSGR: Controls where on the stack we read, and how much the stack pointer moves for
- NSGW: Controls what goes into Write for the stack
- WriteM: This will be set to 1 if we are writing to memory
- PCSrc: Set to 1 if we want to set PC to a value of our choosing
- BNE: Differentiates between Branch Not Equal and Branch Equal
- BComp: Pick either the top of the stack, or 2 lower address below the top of the stack
- Branch: Set to 1 for j, beq, bne. Other logic is still needed to see if we overwrite PC
- ALUOp: Tells ALU what instruction we are doing
- SRS: Reads either SP or SP + offset
- Waitin: 1 if on wait instruction, 0 otherwise; helps with input
- PrintOut: 1 if on ouptput instruction, 0 otherwise; helps with output

Memory:

- Everything in our architecture is block addressable, meaning you can only refer to a 16 bit block, never anything smaller. If you only want to use a bit, for example, it must take up a whole block (the other 15 bits will just be 0s).
- Since we have 16 bit addresses, we can address blocks of memory from 0x0000 to 0xFFFF.
- We have 2 different blocks of memory. One is instruction memory which contains the instructions for the program and also the code for exception handling. The other is data memory which contains data which would need to be kept long term. They both start at address 0 and grow up to higher values.

Memory breakdown

- We have 3 separate "memories"
    1. Instruction Memory
        - This block of memory contains all of the instructions for the programs. It has 15000 blocks.
        - PC points at the current instruction in this memory.
        - This part of memory also contains the exception handling code.
    2. Stack
        - The stack is actually not memory, it is made out of registers.
        - It is the component that stores all of our temporary, short-term use variables and values.
        - It is where we perform the calculations by getting values off the stack and putting the result back on.
        - Stack starts at register 0 and grows up.
        - SP points to the top of the stack.
    3. Data Memory
        - 5000 blocks
        - Used to store long term variables.

        Globals
        - Address: 0x0004-0x0403
        - Global variables and values that can be used across programs.

Registers

- We have a stack of registers for programs, but they can't be referenced by name, and so there are no conventions for their use
- We also have 2 registers that are not part of the stack:
    - sp - our stack pointer; points to the top of the stack
    - pc - points to the current instruction in instruction memory

Stack conventions:

- Whenever you are going to call another function, the following are put on the stack in order:
    - Return address
    - Arguments (in reverse order so that they are in the correct order when popped off)
    - The address of the function you are calling, then you jump to it
- Whenever you are leaving a function, it is assumed that the stack will look the same as it did when entering the current function except for the return value is on top of the stack. So when leaving a function, the following are put on the stack and everything below is assumed to be the same as before:
    - Return value(s)
    - Return address, then jump to the return address

Input/Output

The programmer can use an output instruction whenever they would like to output the value at the top of the stack. If they would like to submit input, they must use a wait instruction.

- Input (wait)

        - The value of input is stored into a register, until the user submits (by pressing the “east” button on the FPGA board) during a wait instruction. As soon as the input is submitted, the value in the input register is written to the top of the stack.
    - Output (output)
        - The value of output is stored into a register during a output instruction, which then outputs (to the FPGA board display)

Example code using I/O

```
        pushi 0             // make space on the stack for input value
        wait                // wait for user to submit value, writes it into the top of the stack
        ouput               // outputs the value at the top of the stack
```

Exceptions

- Handling Exception
    - The only exceptions we will need to handle are Arithmetic Overflow and Stack Overflow. Other exceptions should be handled by the assembler, but we can include, just for the sake of testing, invalid instruction and invalid addresses. When these occur, control notices the exceptions and alters PC and runs the exception handler code.
    - Our ALU will be able to detect Arithmetic Overflow with an overflow bit that flags if there is an overflow.
        - We will tell the user that there is Arithmetic Overflow, but ignore the exception and continue with the instructions.
    - Stack Overflow can be detected by checking if sp will be less than mp when it is updated.
        - We will abort if this happens.

Sample Code:

Euclid’s algorithm		
Assembly		Machine(no waits)
pushli 0	// make space for input "n"	**0001000000000000**
wait	// wait for input	**0111100000000000**
pushli 2	// puts "m" = 2 on stack	**0001000000000002**
relLoop:	// while(gcd(n,m)!=1)	**0011100000000000**
dupo 0	// copy b to top of stack	**0011100000000002**
dupo 2	// copy a to top of stack	**0000100000000000**
pushi gcd	// put label of GCD on stack	**0001000000011110**
j	// jump to gcd, gcd will leave result on top of stack	**0010100000000000**
inRelLoop:		**0100000000000000**
pushli 1	// put 1 on top of stack	**0001000000000001**
pushi relPrimeRet//		**0000100000000000**
beq	// if(gcd(n,m) == 1, then jump to label	**0001000000010111**
dupo 0	// copy m	**0010100000000000**
pushli 1	// put 1 on stack	**0100100000000011**
add	// m+1 on stack	**0011100000000000**
repo 1	// m = m + 1	**0001000000000001**
pop	// remove "m+1" from top	**0010000000000000**
pushi relLoop	//	**0111000000000001**
j	// jump back to while loop	**0000000000000000**
relPrimeRet:	// this will return "m"	**0000100000000000**
repo 1	// copy m to where n was	**0001000000000011**
pop	// pop	**0010100000000000**
output	// outputs to the lcd	**0100000000000000**
pushi exit		**0111000000000001**
j		**0000000000000000**

```
gcd:                    // assuming that args were pushed in reverse order (b, then a)
dupo 0                  // copy "a" to top of stack
pushli 0                  // put 0 on stack to compare
pushi gcdLoop           // label to jump to if "a" != 0
bne                     //
pop                     // we pop "a" off the stack, leaving b on top (as the ret value)
pushi inRelLoop         // push the return address
j                       // jump to relprime
gcdLoop:                // while(b != 0)
dupo 1                  // copy "b" to top of stack
pushli 0                  // put 0 on top of stack
pushi gcdExit           // label to jump to if "b"== 0 (to return "a" and leave gcd)
beq                     //
dupo 1                  // if statement starts here, copy "b" to top of stack
dupo 1                  // copy "a" to top of stack
slt                     // (b < a) ? 1 : 0
pushli 1                  // put 1 on stack
pushi gcdWhileElse      // label of else statement
bne                     // slt != 1? (jump to else) : continue
dupo 0                  // copy a
dupo 2                  // copy b
neg                     // b = -b
add                     // a - b
repo 1                  // a = a + (-b)
pop                     // remove a - b
pushi gcdLoop           // jump to gcdLoop again
j
gcdWhileElse:           // b = b-a
dupo 1                  // copy "a" to top of stack
dupo 1                  // copy "b" to top of stack
neg                     // a = -a
add                     // push "b + (-a)" onto stack
repo 2                  // b = b + (-a)
pop
pushi gcdLoop           // jump to gcdLoop again
j
gcdExit:                // return "a"
repo 1                  // copy value of "a" into "b", since we will pop "a" off
pop                     // pop off the top of the stack, leaves the value of "a"
pushi inRelLoop         //
j                           //
exit:
pop
```

```
1000000000000000
0000100000000000
0001000001010001
0010100000000000
0100000000000000
0011100000000000
0001000000000000
0000100000000000
0001000000101001
0010100000000000
0101000000000011
0000000000000000
0000100000000000
0001000000001001
0010100000000000
0100000000000000
0011100000000001
0001000000000000
0000100000000000
0001000001001011
0010100000000000
0100100000000011
0011100000000001
0011100000000001
0101100000000000
0001000000000001
0000100000000000
0001000001000001
0010100000000000
0101000000000011
0011100000000000
0011100000000002
0011000000000000
0010000000000000
0111000000000001
0000000000000000
0000100000000000
0001000000101001
0010100000000000
0100000000000000
0011100000000001
0011100000000001
0011000000000000
0010000000000000
0111000000000002
0000000000000000
0000100000000000
0001000000101001
0010100000000000
0100000000000000
0111000000000001
0000000000000000
0000100000000000
0001000000001001
0010100000000000
0100000000000000
0000000000000000
```

SAMPLE WHILE LOOP.

```
int a = 0;
int b = 5;
while(a < b){    a = a + 1;

}

          pushi 0        //put a on stack
          pushi 5        //put b on stack
LOOP:     dupo 1         //copy a onto stack
          dupo 1                   //copy b onto stack
          slt            //if a < b, change sp - 1 to 1, else sp - 1 = 0, and pop once
          pushi 0        // put 0 onto stack
          pushi END      //pushes a 16 bit address onto the stack
          beq            // if a < b, then 1 != 0, no not branch. if a !< b then 0 = 0, branch.
                         // pop three times either way ( sp = sp - 3)
          dupo 1         // copy a
          pushi 1        // put 1 on stack
          add            // result = a + 1
          repo 2         //a = result, pop once
          pop
          pushi LOOP               //repeat loop
          j
END:
```

SAMPLE FOR LOOP.

```
int a = 2;
for(int i = 0 ; i < 10; i ++){
          a = a + a;
}

          pushi 2        //put a on stack
          pushi 0        //put i on stack
          pushi 10       //put the condition value on stack
LOOP:     dupo 1         //copy i onto stack
          dupo 1                   //copy condition value onto stack
          slt            // if (i < 10) then (sp - 1)= 1, else (sp - 1) = 0, then pop once (sp= sp - 1)
          pushi 0        // if slt = 0 then i is not < 10
          pushi END      //put addr on stack
          beq            //if (i !< 10)  then 0 = 0, jump to END, otherwise continue
```

```
                    // pop 3 times either way
        dupo 2      // copy a
        dupo 3      // copy a
        add         //result = a + a
        repo 3      //a = result
        pop         //pop once
        dupo 1              //copy i onto stack
        pushi 1     //put 1 on stack
        add         //result = i + 1
        repo 2      // i = result
        pop         //pop once
        pushi LOOP  //put addr of loop onto stack
        j           // jump, pop once
END:
```

SAMPLE IF/ELSE STATEMENTS

```
int a = 10;
int b = 5;
if( a + b  > 15){
    a = b + 10;
}else{
    a = b - 5;
}
```

```
        pushi 10    // put a on stack
        pushi 5     // put b on stack
        pushi 15    // flip the inequality to be (15 < a + b)
        dupo 2      //copy a
        dupo 2      //copy b
        add         // a + b
        slt         //compare 15 < a + b? 1:0
        pushi 0     // we want to branch if 15 !< a + b, so we push on 0
        pushi ELSE  //put addr on stack
        beq         // if( 15 !< a + b)  then 0 = 0 so jump to ELSE
        dupo 0              // copy b
        pushi 10    // put 10 on stack
        add         // result = b + 10
        repo 2      // a = result
        pushi END   //skip over the else statement
        j

ELSE:   dupo 0      // copy b
```

```
          pushi -5          //put -5 on stack
          add               //result = b - 5
          repo 2            //a = result
END:
```