CSSE232 Final Project Report
Fake Stack Machine
Instructor: Michal
Team 2C

Weize Sun, Peicheng Tang, Wenkang Dang, Minzhe Luo

Table of contents

Executive Summary of the design

Our group created a multi-cycle, stack processor that is capable of running Euclid's algorithm. We have 17 instructions and most of them does not use immediates. In our design, we also have a cache to store the temporary data in the stack as working memory so it is fast within the memory access. It costs much few time to run each instruction than any other architecture types. This processor is capable of function calls, recursion and any other programming needs. It is hard to complete a stack processor and we are proud that we made it.

Introduction

Our processor uses a multi-cycle datapath. Most of our instructions use 4 cycles. The processor consists of a program counter, distributing memories, register files, stack, cache, ALU and many muxes to process instructions and programs. The strength of the processor lies in its ability to access memory. Because of its stack architecture and the cache, it makes the processor easy to access to the memory and costs less time to implement a simple instruction. Our original objective was to make the processor easy to understand and to use by the user, but we ended up trying to make the logic clear and make the processor easy on the hardware side to design. We have changed and improved our RTL and instructions to make them more efficient. We have also discarded several instructions that we build initially, since they were not necessary to our design. There are still many places where it could be improved, such as exception handler.

Body

After we decided to take the stack design challenge, coming up with how to write instruction was a lot simpler. Our design implements a multi-channel stack structure which allows a simultaneous write to the stack and a simultaneous output from the stack. The multi-channel input allows fast operation towards "dup", "swap" and "dup2", while the multi-channel output allows fast operation towards all the jump functions. For example, "jpg", which jump on condition if a is greater than b, we put a and b into the ALU while c into the pc, and if the condition is met, we jump, this allowed us to jump without moving the stack pointer. The cache design is another highlight of our design, this is designed to access the "memory" with a single 16-bit instruction.

We made the two controls, "ALUOP" and "ALUi" depend on the first 5 bits of the instruction. This reduced our number of states significantly, and thus allowed us to build a more concise control unit.The ALUi determines what is the data input of the ALU while the ALUop determines the operation of the ALU. The ALUop take the 13:11 bit in the 16-bit instruction while the ALUi take the 14:12 bit in the 16-bit instructions, we altered our instructions accordingly to make this work.

“PCCTRL” only depends on the instruction as well. The PC module takes the first two bits of the instruction to determine whether it is a jump, jump equal to zero or jump if greater than. we altered our instruction in order to put this feature into reality

We added the ALU disable control signal in the end. During debugging, we found out that as soon as we altered the stack pointer, the output from the ALU altered. In order to solve this issue, we decided to disable the ALU instead of changing the stack pointer.

For “IREN”, we figured out that write enable is necessary on the instruction register for multi-cycle, since after we changed the PC, the output data from the memory altered immediately.

The CPU has two components, the datapath and the control. The datapath processes data and manipulates data flow, while the control instructs the datapath what to do. In our multi-cycle processor, the datapath is designed to be synchronous. The Control Unit is a Mealy state machine able to output different control signals in the same state based on the datapath status flag.

The control unit has 16 states in total. All instructions execute in 3 to 4 cycles. For most instructions, there is instruction fetch, decode, memory load, and execution.

Most of our Xilinx models are implemented with Verilog. With Verilog, building a CPU is easy and straightforward.

Muxes can be built with ternary operator in Verilog.

The ALU was implemented with Verilog with cases, which generates a mux.

We used “always@ (posedge CLK)” block in Verilog to implement our Control Unit along with the state numbers to assign the control output. Since the datapath is much slower than the control, and the state transition is not part of the critical path of our CPU, there is no need for optimization. However, our datapath needs to wait for the control signal to start performing its work, so control signals are part of the critical path. Control signals are related to both state and input. Our unit tests are fairly straightforward. For every component that we wrote, we made a list of desired inputs and expected outputs that the corresponding component could provide while it was isolated from the rest of the datapath. In addition to a base case that covered general instructions, each component would have all of its corner and edge cases tested so that our group would know that it could take strange instructions and wouldn’t break seemingly randomly in the middle of a program.

The unit test for control unit requires a lot of different eases. We used the waveform to determining if the state transition is correct and if the control output is what we desired. In order to make the life easier, we try to combine the states and minimize the number of states as few as possible. We have a lot of difficulty testing and fixing the bug of the control unit at the beginning. After checking the waveform patiently, we finally make the control unit pass the unit test. Then in the following integration test, the control unit also perform no more bug.

Integration tests were more difficult. First, we had to come up with meaningful combinations of the units of our datapath. We separated the whole datapath to several sets. Then we listed out all the inputs and outputs of the sets. In order to test if the output of the sets of components is what we expected, we assumed the input and the condition to the combinations. By the waveform that generating from the simulation in hardware, we checked if the actual data were as same as our expected output.

System tests are performed by running the single instruction in the processor. We tested each instruction by put the instruction into the memory. Once each instruction was verified to be working, small sets of instructions were tested together. These test instructions snippets were implement in the milestone 1 of the projects. After the processor passed these test, we finally tested the Euclid's algorithm program on our processor. During the test, we fixed and debugged our processor to ensure that it could come out the result that we expected without problems. Most of the problems we had during making Euclid's algorithm work functionally were the mistakes we made in the hardware design with Xilinx. There was no major mistake about our design logic and processor.

After all these tests were passed, we ensured that our processor works properly.

The performance data of our design is listed below:
1. The total number of bytes required to store both Euclid's algorithm and relPrime as well as any memory variables or constants.

4 variable and a total of 8 bytes
The relPrime use 92 byte program and Euclid's algorithm is 92+4*2=100 bytes

2. The total number instructions executed when relPrime is called with 0x13B0 (the result should be 0x000B using the algorithm specified in the project specifications).

0.26 million instructions

3. The total number of cycles required to execute `relPrime under the same conditions as Step2.

1.04 million cycles
4. The average cycles per instruction based on the data collected in Steps 2 and 3.
4 cycle

5. The cycle time for your design (from the Xilinx Synthesis report – look for the Timing summary).

Minimum input arrival time before clock: 3.561ns
Maximum output required time after clock: 4.368ns
Maximum combinational path delay: 7.041ns

6. The total execution time for relPrime under the same conditions as Step 2.

0.007322640000s = 7.323 ms

7. The gate count for your entire design (from the Xilinx Map report). This appears to changed to cell in recent version.

Cell Usage :
BELS : 24
LUT3 : 16
LUT4 : 8
FlipFlops/Latches : 8
LD : 8
Clock Buffers : 1
BUFG : 1
IO Buffers : 66
IBUF : 42
OBUF : 24

8. The device utilization summary (from the Xilinx Synthesis report).
elected Device : 3s500efg320-4

Number of Slices:	14	out of	4656	0%
Number of 4 input LUTs:	24	out of	9312	0%
Number of IOs:	66			
Number of bonded IOBs:	66	out of	232	28%
IOB Flip Flops:	8			
Number of GCLKs:	1	out of	24	4%

Conclusion

In conclusion, our design came out exactly how we expected it to. We spent lots of time implementing the stack design since there is few example which has similar architecture that we can refer to. Unlike building a MIPS-like load and store processor, we explored a new type of processor that we have not mentioned a lot in class. Our group met frequently to discuss the problem that we need cope with during the design and implementations in the hardware. In the design, we made a cache to provide as a working memory which allowed our processor to access the memory more easily and more quickly. There are still improvements that we would like to add to the processor if we have more time. But there is no major weakness to the design.

Assembly Language Specifications:

This is a stack-based processor which implements a stack machine. All operands are addressed relative to the top-of-stack pointer or a frame pointer, which is use to access local variables. The instructions are 16-bits. The first 5 bits represent op code and last 8-bits represent the immediate.

The instructions are as follows:

D-type

op	X	imm
5	3	8

op: Operation code

X: Unused bits

imm: Constant value or address

This instruction set is used when we need to input or output a value that does not exist in our stack. For example, push A(Put value A at the top of our stack).

L-type

op	X
5	11

op: Operation code

X: Unused 11 bits

This instruction set is used when we only need to manipulate data already in the stack. For example, Add (Add the top two value in the stack and put the result at the top of our stack).

Cache Design:

The cache design is used to buffer the data between the memory and the cpu. While the address of the memory is 16-bits, the address of the cache is designed to be 8 bits so that a 16-bit instruction could access the entire cache within a single operation.

Example of fetch an value in the memory into the cache

Pshi 0x55 //put the immediate 0x55 at the top of the stack

Pshi 0x55 //put the immediate 0x55 at the top of the stack (now the data of first row and the second row of the stack are both 0x55)

SHL 8 //shift the data of the top of the stack left 8 bits.

ADD //add the data of the first two row of the stack and put the result at the top of the stack

M2C 0x0C //The data of the top of the stack is the memory address. This instruction fetches the data of the memory address and save to the cache address 0x0C

Example of store an value from the cache to the memory

Pshi 0x55 //put the immediate 0x55 at the top of the stack

Pshi 0x55 //put the immediate 0x55 at the top of the stack (now the data of first row and the second row of the stack are both 0x55)
SHL 8 //shift the data of the top of the stack left 8 bits.
ADD //add the data of the first two row of the stack and put the result at the top of the stack
C2M 0x0C // This instruction fetches the data at address 0x0C of the cache and save to the memory address which is saved at the top of the stack.

Machine Language Format:

We have 5 bits for the op code.

Instruction set architectureop

Instruction	**Description**	Opcode
ADD	Add the data of first two rows of stack and save it at the top of the stack.	00000
SUB	The data of the top of the stack subtract by data of the second row of the stack and save the result at the top of the stack. (e.g. PSH a PSH b SUB now the data at the top of the stack is b-a)	00001
SHL IMM	Shift the value on the top of the stack to the left by IMM.	00010
SHR IMM	Shift the value on the top of the stack to the right by IMM.	00011
SWP	Swap the top two rows of the stack.	00100
DUP	Duplicate the data at the top stack and store it at the top stack.	00101
DUP2	Duplicate the data of the first two row of the stack and save at the top two row of the stack.	00110
GET	Read value from I/O port and push.	00111
PUT	Write top stack value to I/O port.	01000
DISP	Display the top stack value to LCD.	01001
PSH address	Fetch the data at the given address of the cache and put it on the top of the stack.	10000

	Decrease the stack pointer.	
POP address	Store the data of the top of the stack to the cache at the address. Increase the stack pointer.	10001
JMP	Jump to the address.	11110
JPE	Jump to the address when the data of the first two row of the stack is equal.	10101
JPEZ	Jump when the data of the top stack is equal to 0.	11101
JPG	Jump to the address when the data of the top of the stack is greater than the data of the second row. Decrease the stack pointer.	10100
PSHI immediate	Push the immediate to the top of the stack.	10110
M2C Cache_address	Load the value from the memory to the cache, the address of the memory is the 16-bit value at the top of the stack and the address of the cache is the Cache_address.	10111
C2M Cache_address	Store the value from the cache to the memory, the address of the memory is the 16-bit value at the top of the stack and the address of the cache is the Cache_address.	11000
STP	Stop execution.	11111

Rules of Machine Code Translation

For each instruction, first five bits are the opcode of instruction name. If the opcode is start with 0, the instruction is L-type, otherwise, the instruction is D-type.

For the L-type instruction, the last 11 bits remain 0 for being unused.

For the D-type, the next 3 bits remain 0 for being unused. And the last 8 bits can be the immediate data in binary or the address code. The address codes in the stack memory are set as below.

Address Code of Stack Memory:
Stack[0]=00000000

Stack[1]=00000001
Stack[2]=00000010
Stack[3]=00000011
…...
Stack[n]=eight-bit binary of n

Euclid's Algorithm

Address	Assembly Code	Machine Code	Comment
0x0000	Begin: Get	0011100000000000	Get the input value
0x0001	pop n	1000100000000001	store the value at n
0x0002	pshi 2	1011000000000010	push 2 to the top of stack
0x0003	pop m	1000100000000010	store the value at m
0x0004	pshi Gcd	1011000000010001	push the address of Gcd to the top of stack
0x0005	jmp	1111000000000000	jump to Gcd
0x0006	Done: pshi 1	1011000000000001	push 1 to the top of stack
0x0007	sub	0000100000000000	subtract the first two values and put at the top
0x0008	pshi End	1011000000001111	push the address of End to the top of stack
0x0009	jpez	1110100000000000	jump to End if the top value = 0
0x000a	psh m	1000000000000010	push value stored at m
0x000b	pshi 1	1011000000000001	push 1 to the top of stack
0x000c	add	0000000000000000	second level value add top level value
0x000d	pop m	1000100000000010	store the top value at m
0x000e	pshi Gcd	1011000000010001	push the address of Gcd to the top of stack
0x000f	jmp	1111000000000000	jump to Gcd
0x0010	End: psh m	1000000000000010	push the value stored at m
0x0011	put	0100000000000000	Write the output value
0x0012	Gcd: psh n	1000000000000001	push value stored at n
0x0013	psh m	1000000000000010	push value stored at m
0x0014	psh n	1000000000000001	push value stored at n
0x0015	pop a	1000100000000011	store the top value at a
0x0016	pop b	1000100000000100	store the top value at b
0x0017	pshi Done	1011000000000101	push the address of Done to the top of stack
0x0018	jpez	1110100000000000	jump to Done if the top value=0
0x0019	Loop: psh b	1000000000000100	push value stored at b
0x001a	pshi Finish	1011000000101011	push the address of Finish to the top of stack
0x001b	jpez	1110100000000000	jump to Finish if the top value=0
0x001c	pshi UpdateA	1011000000100101	push the address of UpdateA to the top of stack
0x001d	psh b	1000000000000100	push value stored at b

0x001e	psh a	1000000000000011	push value stored at a
0x001f	jpg	1010000000000000	jump to UpdateA if the top > the second
0x0020	psh b	1000000000000100	push value stored at b
0x0021	psh a	1000000000000011	push value stored at a
0x0022	sub	0000100000000000	subtract the first two values and put at the top
0x0023	pop b	1000100000000100	store the top value at b
0x0024	pshi Loop	1011000000011000	push the address of Loop to the top of stack
0x0025	jmp	1111000000000000	jump to Loop
0x0026	UpdateA: psh a	1000000000000011	push value stored at a
0x0027	psh b	1000000000000100	push value stored at b
0x0028	sub	0000100000000000	subtract the first two values and put at the top
0x0029	pop a	1000100000000011	store the top value at a
0x002a	pshi Loop	1011000000011000	push the address of Loop to the top of stack
0x002b	jmp	1111000000000000	jump to Loop
0x002c	Finish: psh a	1000000000000011	push value stored at a
0x002d	psh Done	1011000000000101	push the address of Done to the top of stack
0x002e	jmp	1111000000000000	jump to Done

Assembly Language Fragments Example Code

Conditional branch:

```
        push a
        push b
loop:   jpg Done
        add1
        jmp loop
Done:   …
```

Load an 16bit value into the stack:

```
        push 0x22
        push 0x22
        SHL 8
        add
```

RTL description of each instruction

Common Cycle 1 for all instructions: IR = Mem[PC]

PC = PC + 1

Cycles	ADD	SUB	SHL IMM	SHR IMM
Cycle 2	a=stack[$sp] b=stack[$sp1] c=stack[$sp2]			
Cycle 3	ALUout = A op B $sp = $sp +2			
Cycle 4	Stack[$sp]=ALUout			
Cycles	SWP	DUP	DUP2	POP address
Cycle 2	a=stack[$sp] b=stack[$sp1] c=stack[$sp2]			
Cycle 3	Stack[$sp]=b Stack[$sp1]=a	$sp++	$sp=$sp+2	cache[IR[7..0]] =a $sp--
Cycle 4		Stack[$sp]=a Stack[$sp1]=a	Stack[$sp]=a Stack[$sp1]=b Stack[$sp2]=a Stack[$sp3]=b	
Cycles	JMP	JPE	JPEZ	JPG
Cycle 2	a=stack[$sp] b=stack[$sp1] c=stack[$sp2]			
Cycle 3	PC =a $sp--	if (a==b) PC=c $sp=$sp-3	if(b==0) PC=a $sp=$sp-2	if(a>b) PC=c $sp=$sp-3
Cycle 4				
Cycles	M2C Cache_address	C2M Cache_address	PSHI immediate	PSH address
Cycle 2	a=stack[$sp]		$sp++	a =cache[IR[7..0]]
Cycle 3	Memout = Mem[a]	Cacheout = Cache[IR[7..0]]	Stack[$sp] =ZE (imm)	Stack[$sp]=a $sp++
Cycle 4	Cache[IR[7..0]]= Memout	Mem[a]=Cacheo ut		
Cycles	GET	PUT		
Cycle 2	Stack[$sp]=Input	Output=Stack[$s p]		

Cycle 3				
Cycle 4				

RTL Testing methods

For all the instructions, we will test the condition that whether they can work properly or not.

Instruction	opcode	testing mode	
ADD	00000	ADD 0000000000000000 Assume Stack[$sp]= 2 and Stack[$sp+1]=3. The expected value of Stack[$sp] after the execution should be 5.	ADD 0000000000000000 Assume Stack[$sp]=0xffffffffffffffff and Stack[$sp+1]=1. The instruction will cause overflow.
SUB	00001	SUB 0001000000000000 Assume Stack[$sp]= 2 and Stack[$sp+1]=1. The expected value of Stack[$sp] after the execution should be 1.	SUB 0001000000000000 Assume Stack[$sp]=1 and Stack[$sp+1]=0xffffffffffffffff. The instruction will cause overflow.
SWP	00100	SWP 0010000000000000 Assume Stack[$sp]= 2 and Stack[$sp+1]=1. The expected value of Stack[$sp] after the execution should be 1 and the expected value of Stack[$sp+1] after the execution should be 2.	SWP 0010000000000000 When the Stack[$sp] or Stack[$sp+1] is empty(has no data in it), this instruction can not work.
DUP	00101	DUP 0010100000000000 Assume Stack[$sp]= 2. The expected value of Stack[$sp] after the execution should be 2 and the expected value of	DUP 0010100000000000 If there is no data at the stack pointer, this instruction can not work.

		Stack[$sp+1] after the execution should be 2	
DUP2	00110	DUP2 0011000000000000 Assume Stack[$sp]= 2 and Stack[$sp+2]=3. The expected value of Stack[$sp] and Stack[$sp+1] after the execution should be 2 and the expected value of Stack[$sp+2] and Stack[$sp+3] after the execution should be 3.	DUP2 0011000000000000 If there is no data at the stack pointer or there is no data at Stack[$sp+2], this instruction can not work.
GET	00111	GET 0011100000000000 Assume the Input port get a value of 1, after GET instruction, the expected value of Stack[$sp] is 1	
PUT	01000	PUT 0100000000000000 Assume the value of Stack[$sp] is 1, after PUT instruction, the expected value of Output port is 1.	
PSH address	10000	PSH 0xFF 1000000011111111 Assume the data of the Cache address 0xFF is 1, the expected value of the Stack[$sp] should be 1	PSH 0xFF 1000000011111111 If there is no data in the Cache address 0xFF, this instruction may cause error.
POP address	10001	POP 0xFF 1000100011111111 Assume the data of the Stack[$sp] is 1, the expected value of the Cache address 0xFF should be 1.	POP 0xFF 1000100011111111 If there is no data in the Stack[$sp], this instruction may cause error.
JMP address	11110	JMP loop 0x0014 loop 1001000000010100 Assume we need to jump to	

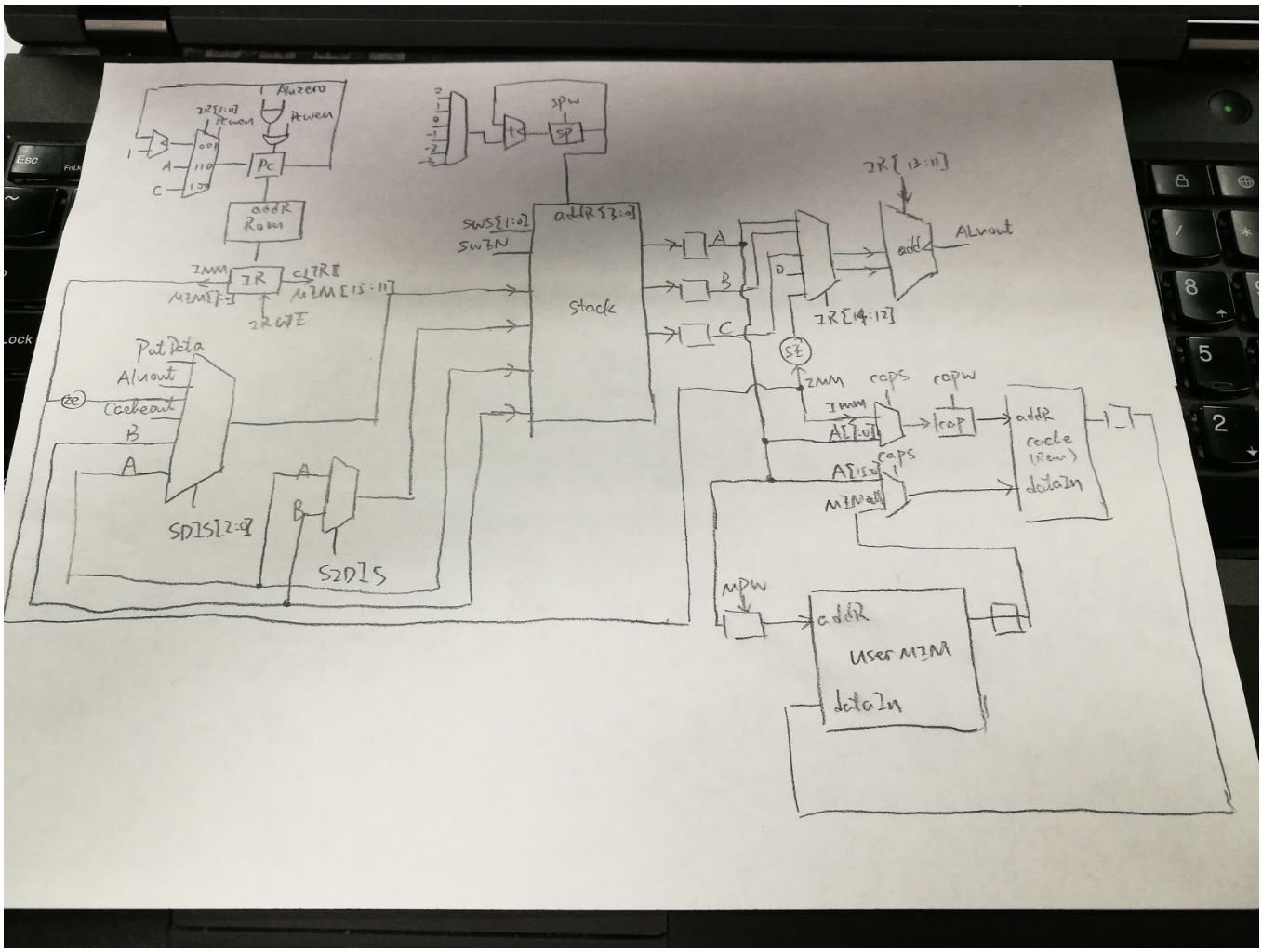
		loop, and the loop address starts at 0x0014. PC then points to 0x0014	
JPE address	10101	JPE loop 0x0014 loop 1001100000010100 Assume at the top of the stack, there are two 2s. Since they are equal, PC points to 0x0014	JPE loop 0x0014 loop 1001100000010100 Assume at the top of the stack, there exists a 3 and a 5. Since 3!=5, PC points to PC+2
JPEZ address	11101	JPEZ loop 0x0014 loop 1010000000010100 Assume at the top of the stack, there exists a 0. Since the top of stack = 0, PC points to 0x0014	JPEZ loop 0x0014 loop 1010000000010100 Assume at the top of the stack, there exists a 1. Since the top of stack != 0, PC points to PC+2
JPG address	10100	JPG loop 0x0014 loop 1010100000010100 Assume at the top of the stack, there exists 8 and 6(8 is above 6). Since 8>6, PC points to 0x0014	JPG loop 0x0014 loop 1010100000010100 Assume at the top of the stack, there exists 6 and 8(6 is above 8). Since 6<8, PC points to PC+2
PSHI immediate	10110	PSHI 1 1011000000000001 After the instruction, the expected value of Stack[$sp] is 1.	
M2C address	10111	M2C 0xFF 1011000011111111 Assume Stack[$sp]=0xFFFF and the data in the Memory address 0xFFFF is 1. The expected value of the Cache address 0xFF after the execution should be 1.	M2C 0xFF 1011000011111111 If there is no data in the Memory address 0xFFFF, this instruction may cause error.
C2M address	11000	C2M 0xFF 1011100011111111 Assume Stack[$sp]=0xFFFF	C2M 0xFF 1011100011111111 If there is no data in the

		and the data in the Cache address 0xFF is 1. The expected value of the Memory address 0xFFFF after the execution should be 1.	Cache address 0xFF, this instruction may cause error.
SHL IMM	00010	SHL 2 1101000000000010 Assume Stack[$sp]= 2. The expected value of Stack[$sp] after the execution should be 8.	
SHR IMM	00011	SHL 2 1101100000000010 Assume Stack[$sp]= 8. The expected value of Stack[$sp] after the execution should be 2.	

List of Components

1. PC(Program Counter)
2. Program Memory(ROM)
3. Control Units
4. Sign Extends Unit
5. Zero Extends Unit
6. Stack
7. Cache
8. User Memory
9. ALUs
10. Muxes

Datapath Design Diagram



Description of Datapath Components

The table below describe the components needed in the datapath and the way to complete them in the hardware Xilinx.

Most of our components are implemented with Verilog, a powerful hardware description language.

Type of component	name	description
Memory/Stack/Cache	User Memory	It is a place to save data. The address of memory is 16-bits. It takes the control bit MWEN (Memory write) so that user can save data temporarily in the user memory. The address of the data target address will be stored in the MP(memory pointer) and can be updated with control signal MPW It's a block memory that could implement by using the example memory block. (Or using block

		memory generator to create one)
	Stack	It acts like a group of 16-bits registers. we can operating data in stack and save data in the stack temporally. Update the stack with control bit SWEN and SWS[1:0] in case SWEN==1 & SWS==00 only stack[sp] will be write in case SWEN==1 & SWS==01 stack[sp-1] and stack[sp] will be write in case SWEN==1 & SWS==10 stack[sp-2] and stack[sp-1] and stack[sp]will be write in case SWEN ==1 & SWS=11 stack[sp-3] and stack [sp-2] and stack[sp-1] and stack[sp] will be write the sp is pointed to the first input channel while sp-1 is pointed to the second input channel, sp-2 pointed to the third input channel and sp-3 pointed to the fourth input channel the three data output channel of this device always output data and do not require control Stack is just a group of 16 16bit registers. To implement this with Verilog, we can directly declare an array of 16 register.
	Cache	It is a place to save temporary data. The address of cache is 8-bits. It takes the control bit CAWR (Cache address write) so that user can save data temporarily in the cache. It's a block memory that could implement by using the example memory block. (Or using block memory generator to create one) cache is just a group of 256 16bit-registers. To implement this with Verilog, we can directly declare an array of 256 register.
	User Memory Pointer	It takes the control bit MPW (memory pointer write) so the user can push and pop. It also take the input of an updated memory pointer that was either incremented or decremented by 1. It outputs the memory pointer.
	Cache Pointer	It takes the control bit CAPW (cache pointer write) so the user can push and pop. It also take the input of an updated cache pointer that was either incremented or decremented by 1. It outputs the cache pointer.

ALU/Adder	ALU	The ALU takes two input A and B and the control bit ALUOP[2:0] which is exactly IR[2:0] from instruction register. The control bit determines what operation the ALU will execute with the inputs ALUop=000 then add is performed ALUop=001 then subtract is performed ALUop=010 then shift left is performed ALUop=011 then shift right is performed ALUop=110 then ALUzero =1 ALUop=101 then if A=B then ALUzero=1 ALUop=100 then if A>B then ALUzero=1 The output of the ALU is ALUout and ALUzero. ALU will be implemented with Verilog with cases. It takes advantage of Verilog's ability to do multibits design. With "case switch", we can describe the output of ALU given specific input, and Verilog magic comes into play and generate corresponded hardware. We can do an overflow check after we have the results. There are still a few means that we can improve our ALU right now. The inputs of the ALU is determined by the IR[13:11] (The middle three bits of the instruction's opcode). case1: 000 When the IR[13:11] is 000, the input of the ALU is A and B. case2: 001 When the IR[13;11] is 001, the input of the ALU is A and signextened immediate. case3: 010 When the IR[13;11] is 000, the input of the ALU is B and 0.
	SPadder	This adder add the sp with const depend on the control signal from SPOP. Since Xilinx implementation can only contain 14bit address, SPadder would ignore SP[11] and SP[10] in addition and always output 0 for them. We can use adder and mux to implement it in Xilinx.
MUX		Muxes make it possible to select the operation and the data we want. these can be easily implemented with case switch statements in Verilog.

Control Signals needed

Name	description
PCWS	select input for the PC
PCWEN	update PC
ALUOP[2:0]	This control signal selects which the operation the ALU will deal with the input.
SDIS[2:0]	This control signal selects which data will be in the stack.
SD2IS[2:0]	This control signal selects which data will be in the under layer of the stack
SPOP[2:0]	It is the stack pointer operation code. This controls the stack pointer movements
SPW	It is the stack pointer write. This controls whether or not to update the stack pointer.
SWS [1:0]	It is the stack write select. This selects how many data lines to be written into the stack.
SWEN	It is the stack write enable. This controls whether or not to write to the stack.
CAPW	It is the cache pointer write. This controls whether or not to write the data to cache pointer.
CAPS	It is the cache pointer select. This selectes the input data/pointer for the cache.
CAWEN	It is the cache write enable. This controls whether or not to write to the cache.
MPW	It is the memory pointer write. This controls whether or not to input address to memory pointer.
MWEN	It is memory write enable. This controls whether or not to write to the memory.
PUT	output the data onto the led
PCJ	enable the PC jump (This control signal need to be set to 1 then Pc can jump)
DISABLE	enable the ALU (This control signal need to set to 1 then ALU can execute)

Testing of Each Component

Type of component	Name	Test
Memory/Stack/Cache	Memory	Write Operation: Case MPW=1; MW_EN=1; The mem pointer will be updated to the value of 'A' (output of the top of the stack).The data from the cacheout will be store to the memory at mem pointer. Case MPW=0; MW_EN=1; The data from the cacheout will be store to the memory at mem pointer. However, the mem pointer will not be updated. Case MPW=1; MW_EN=0; The mem pointer will be updated to the value of 'A' (output of the top of the stack).However, no data will be stored to the memory. Write Operation: MR_EN=1; value of the memory at the mempointer address will be avaliable at Memout.
	PC	The data input of the PC has been already selected by the muxes under the control of the control signals in front of the PC component. PC will change to the input data.
	Stack	Component test: STACK WRITE Case: SWS=00; SWEN=1; the top stack will be write Case:

		SWS=01;
		SWEN=1;
		the top two stack will be write
		Case :
		SWS=10;
		SWEN=1;
		the top three stack will be write
		Case;
		SWS=11;
		SWEN=1;
		the top four stack will be write
		Case:
		SWEN=0;
		SWS=XX;
		No stack will be write
		
		Stack Read:
		Case:
		SWEN=1
		SWS=00
		The top stack will be Read
		SWEN=1
		SWS=01
		The top two stack will be Read
		SWEN= 1
		SWS=10
		The top three stack will be Read
		SWEN=0;
		SWS=xx
		No Stack will be read
		
		Integeration test:for the instruction DUP
		case
		SWS=00;//select top of stack as avaliable input
		SPOP=100;//sp+1
		SPW=1;//stack pointer write enable
		SDIS=000;//select data to the top of the stack
		SWEN=1;//enable data write to the stack
		select the top of stack as input channel, the stack pointer will increase by one. Data A was selected to be the input data and the data will be write to the top of the stack.
		
		Integeration test:for the instruction add, when the control signal has been set:
		PCW=0//pc+2
		SWS=01//select read A,B

		SWEN=1//read enable
		ALUOP=000//select alu operation as add
		SDIS=100 //select stack input as ALUOUT
		SWEN=1//enable stack write
		SWS=00//select top stack as input
		SPOP=010//remove 1 stack
		SPW=1//stack pointer write enable
		
		and the data at the top of the stack is 1 and the data at the second row of the stack is 2, the data at the top of the stack after the execution of the instruction is 3.

	Cache	Component test Cache Write: Case: CAPW=1 CAWEN=1 CAPS=1 the cache pointer will be updated to the value from the immediate and the data from A(top of the stack) will be write to the cache at the cache pointer address. Case: CAPW=1 CAWEN=1 CAPS=0 the cache pointer will be updated to the value from from A[7:0](last 8 bits of the top of the stack) and the data from MEMOUT will be write to the cache at the cache pointer Case: CAWEN=0 CAPS=X CAPW=X No cache write operation will be performed. Intergration test for the cache (perform like PSHI IMM) PCW=0//pc+2

		CAPW=1//cache pointer write enable CAREN=1//cache read enable SPOP=100//sp+1 SPW=1//sp write enable SDIS=010//select cach out as data input for stack SWS=00//select top of stack as input SWEN=1//stack write enable Read the cache at the address of the immediate to the top of the stack. we first select the immediate as the stack pointer input source, and then we enable write to the stack pointer. hence we read the cache from the given stack pointer address. meanwhile, we will reserve a stack by moving the stack pointer upward (sp+1) for the incoming data. Then we select cacheout as our data input for the stack and enable the stack write for the top stack. The expected result : the data on the stack at given address will be write to the top of the stack
ALU/Adder	ALU	ALU test: add constiution: case1: Set the following: A=0x01; B=0x02; op = (000) Output: ALUout= 0x03 case2: Set the following: A=0x01; B=0xff; op = (000) Output: ALUout = 0x00 Overflow =1
		subtract constitution: case1: Set the following: A=0x02; B=0x01; op = (001) Output: ALUout = 0x01 case2: Set the following: A=0xfe; B=0xfc; op = (001) Output: ALUout = 0x02 Overflow=1
		equal constiution: case1: Set the following: A=0x02; B=0x01; op = (010) Output: ALUout = 0x00

		case2: Set the following: A=0x01; B=0x01; op = (010) Output: ALUout = 0x01
		larger than case1: Set the following: A=0x02; B=0x01; op = (011) Output: ALUout = 0x01 case2: Set the following: A=0x02; B=0x02; op = (011) Output: ALUout = 0x00
		shift right case1: Set the following: A=0x04; B=0x02; op = (100) Output: ALUout = 0x01 case2: Set the following: A=0x40; B=0x03; op = (100) Output: ALUout = 0x05
		shift left case1: Set the following: A=0x02; B=0x02; op = (101) Output: ALUout = 0x08 case2: Set the following: A=0x05; B=0x03; op = (101) Output: ALUout = 0x40
MUX	PCWEN and IR[1:0] (which combined as a [2:0] control singal) mux	When the combined control signal is 001 (IR[1:0] = 00, PCWEN = 1), the mux should select PC+1; When the combined control signal is 110 (IR[1:0] = 11, PCWEN = 0), the mux should select the data in the top of the stack; When the combined control signal is 100 (IR[1:0] = 10, PCWEN = 0), the mux should select the data at the third layer of the stack.
	SPOP [1:5]mux	When the control signal is 0, the mux selects the value -3; when the control signal is 1, the mux selects the value -2; when the control signal is 2, the mux selects the

		value -1; when the control signal is 3, the mux selects the value 0; when the control signal is 4, the mux selects the value 1; when the control signal is 5, the mux selects the value 2.
	CAPS mux	When the control signal CAPS is set to 0, the mux should select the A[7:0] (the last eight bits of the data at the top of the stack); when the control signal CAPS is set to 1, the mux should select the immediate..
	S2DIS mux	When the control signal S2DIS is set to 0, the mux should select the value of stack[sp1]; when the control signal S2DIS is set to 1, the mux should select the value of stack[sp].
	SDIS[4;0] mux	When the control signal SIDIS is set to 0, the mux should select the value of stack[sp]; When the control siganl SIDIS is set to 1, the mux should select the value of stack [sp1]; when the control signal is set to 2. the mux should select the value of the output of the cache; when the control signal is set to 3 , the mux should select the immediate; when the control signal is set to 4, the mux should select the output of the ALU; when the control signal is set to 5, the mux should select the input data.

Intergration Plan

We plan to test the datapath separately by divide the whole datapath to several parts. We will test the input and output of the set of the components by its logic.

For each set of the components, we will give out the initial condition and the inputs. We will check if the actual output is as the same as the expected output. If the output is not what we expected, we will try to fix the logic.

Intergration Test according to the Plan

Component	Input	Output	Test

PC+Cache	**MEM[PC]**	**CacheOut**	PCW=0
			//pc=pc+2
			CAPIS=0
			//select IMM as address input
			//assume IMM =0x00
			CAPW=1
			//update the address
			//the address of the //cache is set to //0x00
			CAREN=1
			//read the cache
			//assume //cache[sp]=0x0000
			//data on the cacheout should be 0x0000

stack+cache +memory	**Mem[pc]**	**Cacheout Memoryout**	**Assume stack[sp]=0x0031**
			Assume IMM =0x01
			Assume UserMem[0x0031]=0x0011
			PCW=0//pc+2
			SREN=1//stack read enable
			SRS=00//select top of the stack //A=0x0031
			MPW=1//memory pointer write //write 0x0031 to the memory pointer
			MREN=1//enable memory read
			Memout=UserMEM[0x0031]
			Memout =0x0011
			CAPIS=0//select IMM as cache pointer input
			CAPW=1//update the cache pointer, write 0x01 to the pointer
			CAPS=0//cache data in select
			CAWEN=1//write the cache with memory out= 0x0011
			CAREN=1//Read the cache and now CacheOut =0x0011

PC+Mem	**instruction**	**MEM[7:0]**	case1:

		MEM[15:11]	the input instruction is 0000000000000000, The expected output Mem[7:0] is 0000000 and the expected output Mem[15:11] is 00000. case2: the input instruction is 0000100000000000. The expected output Mem[7:0] is 0000000 and the expected output Mem[15:11] is 00001.

PC+Stack	**MEM[PC]**	**A**	Assume IMM= 0x01
			PCW=0//PC+2
			SPOP=100//sp+1
			SPW=1//write SP
			SDIS=011//select IMM as input for stack
			SWS=00//select top of stack as input
			SWEN=1// write the stack
			//stack[sp]= IMM=0x01
			SRS=00//select the top of the as output
			SREN=1//read the top of stack
			Now A=0x01

Stack+ALU	**A, B, C** **ALUi[1:0]** **ALUop[2;0]**	**ALUout** **ALUzero**	case1: input: A=0x01, B=0x02, C=0x03, ALUi=000, Aluop=000, Expected output: ALUout=0x03 ALUzero=0 case2: input: A=0x03, B=0x02, C=0x03, ALUi=000, Aluop=001, Expected output: ALUout=0x01 ALUzero=0 case3: input: A=0x01, B=0x02, C=0x03, ALUi=001, Aluop=010,

			Expected output: ALUout=0x04 ALUzero=0
IR+Stack	**Mem[7:0] Mem[15:11]**	**A,B,C**	case1: The input Mem[7:0] is 0000000 and the Mem[15:11] is 00000. The initial value of A is 0x01 and B is 0x02 and C is 0x00. After the execution, the expected value at A is 0x03 and B is 0x02 and C is 0x00. case2: The input Mem[7:0] is 0000000 and the Mem[15:11] is 00001. The initial value of A is 0x02 and B is 0x02 and C is 0x00. After the execution, the expected value at A is 0x00 and B is 0x02 and C is 0x00.
Stack+pc+ Mux+memory	**instruction**	**A,B,C**	case1: input instruction is 1011000000000001. Assume there is no value in the stack. After the execution, the expected value in output A is 0x01, The output B and C should have no value. case2: input instruction is 000000000000000. assume the initial value at the top of stack is 0x01 and the value at the second layer of the stack is 0x02. After the execution, the expected value in output A is 0x03. The output B is 0x02 and C should have no value.

Control signals for RTL

The following table list the control signals that need for each cycle of operation of each instruction. The state machine is based on this table.

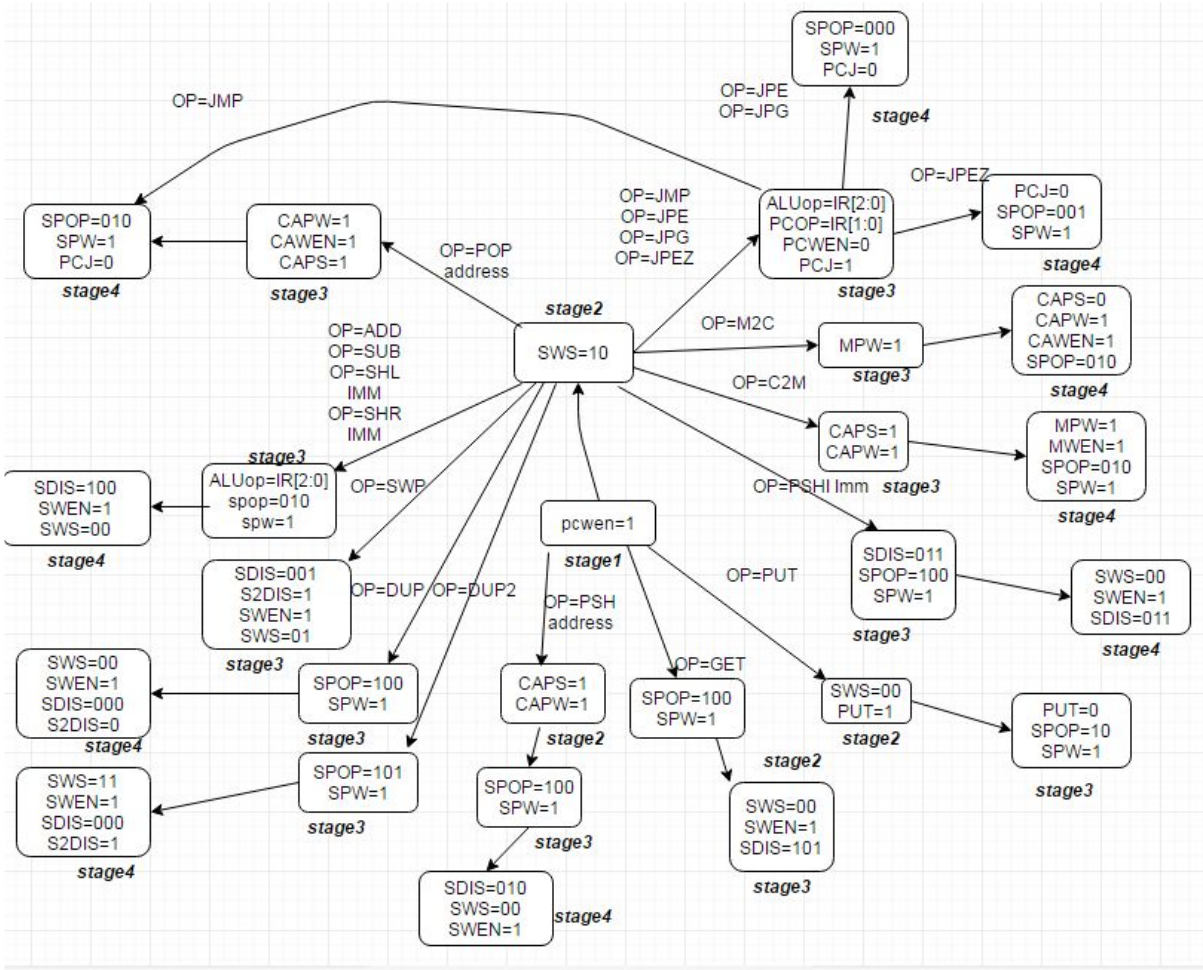
Cycles	ADD	SUB	SHL IMM

Cycle1	PCWEN=1		
Cycle2	SWS=10		
Cycle3	ALUOP=IR[2:0] SPOP=010 SPW=1		
Cycle4	SDIS=100 SWEN=1 SWS=00		
	SHR IMM	SWP	DUP
Cycle1	PCWEN=1		
Cycle2	SWS=10		
Cycle3	ALUOP=IR[2:0] SPOP=010 SPW=1	SDIS=001 S2DIS=1 SWEN=1 SWS=01	SPOP=100 SPW=1
Cycle4	SDIS=100 SWEN=1 SWS=00 SWS=00		SWS=00 SWEN=1 SDIS=010 S2DIS=1
	DUP2	GET	PUT
Cycle1	PCW=0 PCWEN=1		
Cycle2	SWS=01	SWS=00 SWEN=1	SWS=00
Cycle3	SPOP=101 SPW=1	SPOP=100 SPW=1	PUT=1
Cycle4	SWS=11 SWEN=1 SDIS=000 S2DIS=1	SWS=00 SWEN=1 SDIS=101	PUT=0 SPOP=10 SPW=1
	PSH address	POP address	JMP address

Cycle1	PCW=0 PCWEN=1		
Cycle2	CAPSS=1 CAPW=1	SWS=10	SWS=10
Cycle3	SPOP=100 SPW=1	CAPW=1 CAWEN=1 CAPS=1	ALUOP=IR[2:0] PCOP=IR[1:0] PCWEN=0 PCJ=1
Cycle4	SDIS=010 SWS=00 SWEN=1	SPOP=010 SPW=1 PCJ=0	
	JPE address	JPEZ address	JPG address
Cycle1	PCW=0 PCWEN=1		
Cycle2	SWS=10		
Cycle3	ALUOP=IR[2:0] PCOP=IR[1:0] PCWEN=0 PCJ=1		
Cycle4	SPOP=000 SPW=1 PCJ=0	SPOP=001 SPW=1 PCJ=0	SPOP=000 SPW=1 PCJ=0
	PSHI immediate	M2C Cache_address	C2M Cache_address
Cycle1	PCW=0 PCWEN=1		
Cycle2	SWS=10		
Cycle3	SDIS=011 SPOP=100 SPW=1	MPW=1	CAPS=1 CAPW=1
Cycle4	SDIS=011 SWS=00 SWEN=1	CAPS=0 CAPW=1 CAWEN=1	MPW=1 MWEN=1 SPOP=010

		SPOP=010	SPW=1

Graph of the State Machine “Bubbles”



Control Signal output

The table below lists out the wanted control signal for each instruction at each state.
The state of the instruction is determined by the opcode and the cycle of the instruction.

state	opcode	instruction	output
state 1:	**xxxxx**	**all instructions**	**PCWEN=1**
state 2:	**00000** **00100** **00001** **00101** **00110** **00010** **00011**	**ADD** **SWP** **SUB** **DUP** **DUP2** **SHL imm** **SHR imm**	**SWS=10**

	10001 **10101** **10100** **11110** **11101** **10110** **10111** **11000**	**POP address** **JPE** **JPG** **JMP** **JEPZ** **PSHI Imm** **M2C address** **C2M address**	
	01000	**PUT**	**SWS=00** **PUT=1**
	00111	**GET**	**SPOP=100** **SPW=1**
	10000	**PSH address**	**CAPS=1** **CAPW=1**
state 3:	**00000** **00001** **00010** **00011**	**ADD** **SUB** **SHL imm** **SHR imm**	**ALUop=IR[2:0]** **SPOP=010** **SPW=1**
	00100	**SWP**	**SDIS=001** **S2DIS=1** **SWEN=1** **SWS=01**
	00101 **00111** **10110** **10000**	**DUP** **GET** **PSHI imm** **PSH address**	**SPOP=100** **SPW=1**
	00110	**DUP2**	**SPOP=101** **SPW=1**
	01000	**PUT**	**PUT=0** **SPOP=10** **SPW=1**
	11000	**C2M address**	**CAPS=1** **CAPW=1**
	10111	**M2C address**	**MPW=1**
	11110 **10101** **10100** **11101**	**JMP** **JPE** **JPG** **JPEZ**	**ALUop=IR[2:0]** **PCOP=IR[1:0]** **PCWEN=0** **PCJ=1**

	10001	**POP address**	**CAPW=1** **CAWEN=1** **CAPS=1**
state 4:	**00000** **00001** **00010** **00011**	**ADD** **SUB** **SHL imm** **SHR imm**	**SDIS=100** **SWEN=1** **SWS=00**
	00101	**DUP**	**SWS=00** **SWEN=1** **SDIS=000** **S2DIS=0**
	00110	**DUP2**	**SWS=11** **SWEN=1** **SDIS=000** **S2DIS=1**
	10000	**PSH address**	**SDIS=010** **SWS=00** **SWEN=1**
	10110	**PSHI imm**	**SDIS=011** **SWS=00** **SWEN=1**
	11000	**C2M address**	**MPW=1** **MWEN=1** **SPOP=010** **SPW=1**
	10111	**M2C address**	**CAPS=0** **CAPW=1** **CAWEN=1** **SPOP=010**
	11101	**JPEZ**	**SPOP=010** **SPW=1** **PCJ=0**
	10101 **10100**	**JPE** **JPG**	**SPOP=000** **SPW=1** **PCJ=0**
	11110 **10001**	**JMP** **POP address**	**SPOP=010** **SPW=1** **PCJ=0**

Control Unit Xilinx implementation design

The Control Unit will be a onehot moore state machine. As long as we have the state diagram we can directly implement it. Our Control Unit will be a moore machine, so our output most depends on the state.

Each instruction starts to do different work at cycle 2. The number of cycles can be reduced further. In our project, we implement one large control units,

It is complex because each set for each instruction is different from the other and we have to find the optimal gate design manually.

Control Unit Testing

Control unit testing will involve running all combinations of Op and Perform in the state diagram and checking its control signal output and state transition in the above table.

All cases will be tested exclusively. However, it will be tedious to test every case, so we created another module called ControlUnitTester to help us.

ControlUnitTester module takes state, input to controls, and outputs corresponding control signals and the next state. For control signals that don't matter, ControlUnitTester will output an extra bit to specify whether this signal is necessary. ControlUnitTester is implemented simply by copying each in FSM.

ControlUnitTester computes answers rather than storing them. In the test bench, we can run ControlUnit with a specified input combination for multiple cycles and check the output and transition to ControlUnitTester. The checking also covers the case of where the control signal doesn't matter.

In actual code, for example, SREN is the output from ControlUnit, and is compared with the standard answer SREN from ControlUnitTester. If they equal or SREN indicates "don't care", then SREN is 1 to indicate that SREN passes the test in this case. If any control signal does not pass the test, "error" will be high. Sometimes, our FSM will transmit to some impossible state, with all control signals set to "don't care". We do not want to miss those errors, so ControlUnitTester will also output an error signal.

System test

in order to test the whole processor and datapath, we use several instructions to test the system and we decide it to run several small set of instructions.

We simply give the input and instructions to see if the output is same as what we expected so that we can determine if the system behavior properly.

test instruction	control signal needed	initial condition	after execution
PUSHi 1	pcws=0 pcwen=1	at the beginning of the test, the value at	After the execution the expected value

	sws=10 spop=100 spw=1 sdis=011 sws=00 swen=1	the top of the stack is 0.	at the top of the stack is 1.
PUSHi 2	pcws=0 pcwen=1 sws=10 spop=100 spw=1 sdis=011 sws=00 swen=1	the value at the top of the stack is 1 and the other layer of the stack has no value.	after the execution, the expected value at the top of the stack is 2 and the expected value of second layer of the stack is 1.
ADD	pcws=0 pcwen=1 sws=10 ALUop=IR[2:0] spop=010 spw=1 sdis=100 swen=1 sws=00	the value at the top of the stack is 2 and the value at the second layer of the stack is 1.	after the execution, the expected value at the top of the stack is 3 and the expected value of second layer of the stack is 1.
SUB	pcws=0 pcwen=1 sws=10 ALUop=IR[2:0] spop=010 spw=1 sdis=100 swen=1 sws=00	the value at the top of the stack is 3 and the value at the second layer of the stack is 1.	after the execution, the expected value at the top of the stack is 2 and the expected value of second layer of the stack is 1.

To test our full datapath, we will use instructions that have been previously specified in this document. In order to test individual instructions, we will use the tests described in the RTL testing methods; to test smaller groups of instructions, we will use the groups written in the Assembly Language Fragments (like the table above, which is a simple example) for difficult instructions. Once all of those previous tests are passed, we will test Euclid's algorithm. Because Xilinx takes time to load .coe files and these files need to initialize memory, we have decided to add muxes that allow for user input into our memory and register files to save time.

Also, to reduce repetitive code blocks in Verilog, we decided to use Verilog tasks to

perform common operations like loading instructions into memory and specifying initial register file instructions.

Performance of the Design

1. The total number of bytes required to store both Euclid's algorithm and relPrime as well as any memory variables or constants.

4 variable and a total of 8 bytes

2. The total number instructions executed when relPrime is called with 0x13B0 (the result should be 0x000B using the algorithm specified in the project specifications).

0.26 million instructions

3. The total number of cycles required to execute `relPrime under the same conditions as Step2.

1.04 million cycles
4. The average cycles per instruction based on the data collected in Steps 2 and 3.
4 cycle

5. The cycle time for your design (from the Xilinx Synthesis report – look for the Timing summary).

Minimum input arrival time before clock: 3.561ns
Maximum output required time after clock: 4.368ns
Maximum combinational path delay: 7.041ns

6. The total execution time for relPrime under the same conditions as Step 2.

0.007322640000s = 7.323 ms

7. The gate count for your entire design (from the Xilinx Map report). This appears to changed to cell in recent version.

Cell Usage :
BELS : 24

```
#      LUT3                 : 16
#      LUT4                 : 8
# FlipFlops/Latches         : 8
#      LD                   : 8
# Clock Buffers             : 1
#      BUFG                 : 1
# IO Buffers                : 66
#      IBUF                 : 42
#      OBUF                 : 24
```

8. The device utilization summary (from the Xilinx Synthesis report).
elected Device : 3s500efg320-4

Number of Slices:	14	out of	4656	0%
Number of 4 input LUTs:	24	out of	9312	0%
Number of IOs:	66			
Number of bonded IOBs:	66	out of	232	28%
IOB Flip Flops:	8			
Number of GCLKs:	1	out of	24	4%