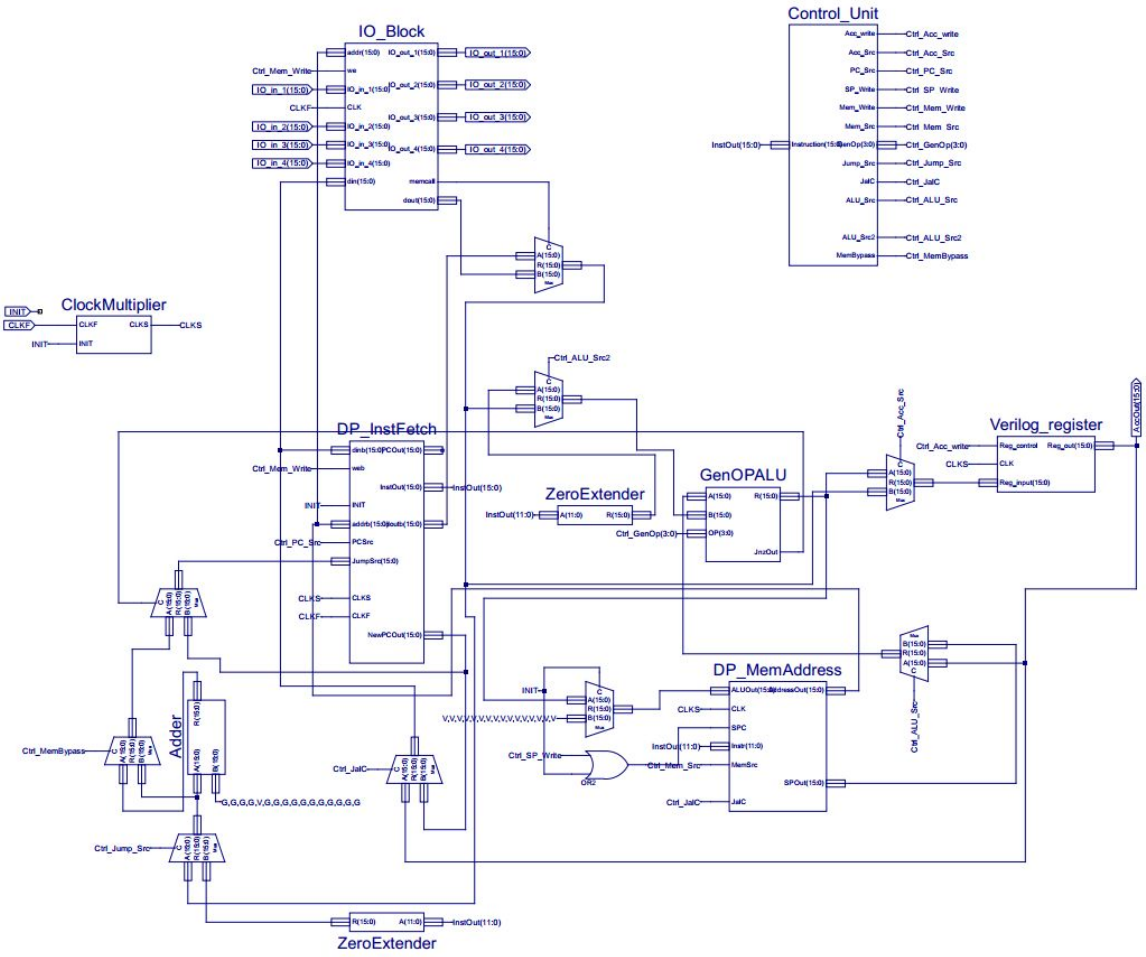


Team 3T Final Project:

General Purpose Processor

Accumulator Design Architecture

CSSE 232 - Computer Architecture 1

Designers:

Tyler Reinhardt

Jason Duncan

Calvin Weaver

Josh Osborne

Table of Contents

*Important note, in PDF format, the table of contents is not accurate due to extra space being added during the file type conversion.

Overview

Our team designed a general purpose processor that contains all instructions and datapath functions necessary to run an assembly-based implementation of Euclid's Algorithm. This processor makes use of a custom assembly language that features original instructions and an extremely small instruction profile. This decision to use a minimal instruction profile combined with our accumulator-based architecture design helps to create an extremely small datapath footprint.

Our processor is made up of numerous custom components including but not limited to:

- A 16-bit central accumulator register
- A 16-bit central stack pointer register
- A 16-bit central program counter register and program counter loop module
- A 10-bit dual-port main memory block and instruction fetching module
- A 16-bit general operations ALU
- An 11-bit sign extender
- An 11-bit zero extender
- A 16-bit MUX
- A 16-bit general adder
- An 16-bit memory address scaler

Our primary objectives were to achieve flexibility of usage, flexibility of implementation and minimal physical size. Our custom assembler allows users to quickly learn how to write programs and produce memory initialization files for our processor. Our memory module is interfaced through our address scalers, a design choice that allows users to swap the main memory block with memory blocks of larger (or smaller) sizes, depending on the user's requirements. Our current implementation exemplifies this, as our instruction set is designed to handle 16-bit memory addresses but our memory module is limited to only 10-bit addresses. Lastly, our instruction set uses a custom format that allows references to both stack-relative and global variable memory to be used interchangeably next to a single opcode. This permits users to, in the same program, manipulate data in both the stack and global variable space without any performance degradation.

Designing the Language

Our processor utilizes a pure accumulator-based architecture. Choosing an accumulator architecture gives us ample room for creative additions in the future. Without temporary registers our instruction set is limited to a small number of very simple logic operations that all manage data relative to the accumulator register. This reduces our instruction profile drastically. In order to fit conditional statements into this streamlined profile, we were required to add instructions that support a multi-instruction conditional structure that "set[s]" the accumulator value to one under various conditions (and otherwise "clear[s]" it to zero). These conditionals are all relative to the value already in the accumulator and the provided instruction argument. For convenience, we added checks that behave inversely to each other (and to eliminate potential instructions and reduce our code profile, at the expense of another instruction in the instruction set)(for example, **`seq`** [Set if EQual] versus **`sne`** (Set if Not Equal).

In the absence of temporary registers, we needed a space to easily store data for multi-variable operations and nested procedure data communication. For this we chose to use the stack. We added a custom instruction, **`sbr`** (Stack Block Reservation), that manipulates the stack pointer to allow for better management of procedure data.

Terminology:
"acc." stands for accumulator

"`$a`" is the accumulators' register
"`arg`" is the immediate value in each instruction
"`$sp`" is the stack pointer register
"`$ra`" is the return address (always stored in the stack at `0($sp)`)

OP	Name	Fmt.	Desc.	Pseudo
0000	**add**	M	add value @ address to value in acc.	$a = $a + mem(arg);
0001	**addi**	L	add immediate to value in acc.	$a = $a + arg;
0010	**sub**	M	subtract value @ address from value in acc.	$a = $a - mem(arg);
0011	**li**	L	set acc. to immediate	$a = arg;
0100	**lw**	M	load value @ address to acc.	$a = mem(arg);
0101	**sw**	M	save value in acc. to arg	mem(arg) = $a;

0110	**seq**	M	set acc. if acc. is equal to value @ arg	if($a == mem(arg)) $a = 1; else $a = 0;
0111	**sne**	M	set acc. if acc. is not equal to value @ arg	if($a == mem(arg)) $a = 0; else $a = 1;
1000	**sgt**	M	set acc. if acc. is greater than value @ arg	if($a > mem(arg)) $a = 1; else $a = 0;
1001	**slt**	M	set acc. if acc. is less than the value @ arg	if($a < mem(arg)) $a = 1; else $a = 0;
1010	**j**	L	jump to address	PC = arg + 0x0800;
1011	**jal**	L	jump and link to an address	$ra = PC; PC = arg + 0x0800;
1100	**jnz**	L	jump to address if acc. is not zero	if($a =! 0) PC = arg + 0x0800;
1101	**ret**	L	jump to return address	PC = $ra;
1110	**sbr**	L	move stack pointer by arg amount 2	$sp = $sp + arg;

Example Code Snippets

Editing a value in memory (where `A` is a memory address):

```
# Increasing the value at memory location A by 8
```

lw A	# Load the value at A into acc.	*0100 011000000000*
addi 8	# Add immediate 8 to acc.	*0001 000000001000*
sw A	# Save acc. value to memory location A.	*0101 011000000000*

```
//Equivalent C code:

A += 8;
```

Running a conditional statement (where `A` is a memory address):

```
# Jump to RUN if A is equal to 16
```

li 16	# Put immediate 16 in acc.	*0011 000000010000*
seq A	# Set acc. to 1 if A is equal to # value in acc.	*0110 011000000000*
jnz RUN	# Jump to RUN if acc. is not equal # to zero	*1100 000000100000*
addi 0	# NOOP so jump can complete	*0001 000000000000*
...		
RUN:	# RUN is located at 0x0820 # conditional code goes here…	

```
//Equivalent C code:

if(A == 16){
      //conditional code goes here...
```

```
}
```

Running procedure and returning:

```
# Run procedure PROC and then jump back
```

sbr -1	# Reserve a word on the stack for ra	*1110 111111111111*
jal PROC	# Jump-and-link to PROC	*1011 000000100000*
addi 0	# NOOP so jump can complete	*0001 000000000000*
sbr 1	# Restore used stack space	*1110 000000000001*
# post-procedure code goes here...		
PROC:	# PROC is located at 0x0820 # procedure code goes here...	
ret	# Return to the PROC caller	*1101 XXXXXXXXXXXX*
addi 0	# NOOP so jump can complete	*0001 000000000000*

```
//Equivalent C code:

void proc(){
     //function code goes here...
}
proc();
//post-function code goes here…
```

Running procedure with arguments and returning a value:

```
# Run procedure SUM and then jump back
```

sbr -4	# Reserve space on the stack for 2 # args, a return value and ra	*1110 111111111100*
li 7	# Put immediate 7 into acc.	*0011 000000000111*
sw 1($sp)	# Save acc. value into the first arg. # stack	*0101 100000000001*
li 8	# Put immediate 8 into acc.	*0011 000000001000*
sw 2($sp)	# Save acc. value into the second arg. # stack	*0101 100000000010*
jal SUM	# Jump-and-link to SUM	*1011 000000100000*
addi 0	# NOOP so jump can complete	*0001 000000000000*
# 6($sp) is now 15, return value operations here		
sbr 4	# Restore used stack space (after using the returned value)	*1110 000000000100*
SUM:	# SUM is located at 0x0820	
lw 1($sp)	# Load first procedure arg. into acc.	*0100 100000000001*
add 2($sp)	# Add first procedure arg. to acc.	*0000 100000000010*
sw 3($sp)	# Save acc. value into first procedure # return value stack space	*0101 100000000011*
ret	# Return to the SUM caller	*1101 XXXXXXXXXXXX*
addi 0	# NOOP so jump can complete	*0001 000000000000*

```
//Equivalent C code:

void sum(int a, int b){
     return a + b;
}
int r = sum(7, 8); //r now has the value of 15
```

Instruction Format

Our instructions have two different instruction formats

L-Format	
OpCode	Immediate
4 bits	12 bits

The first instruction format (the “logical” format) is reserved for logic and non-memory operations. This format includes 4 bits for the instruction opcode, and 12 bits for an immediate value. This is a generic immediate-based format and there are no special functions involving this format. There are few enough instructions that this format shares opcodes with the next format.

M-Format		
Opcode	Special	Immediate
4 bits	1 bit	11 bits

The second instruction format (the “memory” format) is reserved for memory-based operations. This format includes a special bit that is used to determine how the immediate should be translated into a 16 bit memory address. If this bit is 1 then the immediate is treated as a stack-pointer-relative address, where the immediate value is (as an 11 bit unsigned value) added directly to the stack pointer address. If the special bit is 0, then the immediate is interpreted as a memory address, with the remaining most significant bits treated as 0s. This limits the addressable locations in memory to anything less than 2^11 memory slots above the stack pointer (in the case of 1 special bit), or any location below 0x0800 (in the case of 0 special bit). As a consequence, memory allocations are designed to divide up the space below 0x0800 as evenly as possible.

The assembler will automatically determine the value of the special bit based on the formatting of the instruction arguments. For example, instruction arguments with “x($sp)” will generate a 1 special bit value, and instruction arguments with a variable reference will automatically generate a 0 special bit value.

Memory Allocation

Theoretical (16-bit addresses)	Implementation (10-bit addresses)
Range: 0x1800 - 0xFFFF Stack	Range: 0x200 - 0x3FF Stack
Range: 0x0800 - 0x17FF Code	Range: 0x080 - 0x1FF Code
Range: 0x0600 - 0x07FF Global Variables	Range: 0x040 - 0x07F Global Variables
Range: 0x0500 - 0x05FF I/O	Range: 0x020 - 0x03F I/O
Range: 0x0000 - 0x04FF OS	Range: 0x000 - 0x01F OS

The stack begins at 0xFFFF and descends as it grows larger. Back-referencing in the stack requires adding a value to the stack pointer (and thus traveling up in memory address space). Code begins at 0x0800 and ascends with increasing code length. OS instructions start at 0x0000 and also ascend. Our implementation code is downscaled to make use of the available 10-bit address space.

Procedure Call Conventions

Our procedures make use of data stored in the stack. By convention, each program that uses a procedure call must prepare the stack before the procedure call (allocating space and assigning arguments) and clean up after the procedure call (this is *not* the job of the procedure itself!). Procedures may reserve space on the stack for their own use (or their own procedure calls), however space needed for return addresses, procedure arguments and return values must be reserved by the caller.

In order to reserve stack space, the **`sbr`** instruction should be used to move the stack pointer an appropriate amount (i.e. **`sbr -16`** reserves 16 words on the stack for use by a procedure). At least one word must be reserved on the stack for procedures that include a **`ret`** call so that the return address can be stored. The first space on the stack is always reserved for the return address. Spaces immediately following the return address are always reserved for procedure arguments. Spaces immediately following arguments are always reserved for return values. A visual representation of this format is as follows:

Return Address	0($sp)
Arguments	x($sp)
Return Values	x+x2($sp)

All space reserved on the stack must be removed after use. Stack space can be used for operations other than procedure calls, although it should eventually be removed (following traditional conventions). Arguments are not necessarily preserved through procedure calls, and return values are (obviously) overwritten after a procedure call. Procedures are not limited to a fixed number of arguments or return values.

Available Registers

As we are using accumulator architecture, there are few available registers. Almost all data is either stored on the stack, or in memory as variables. The two available registers are:

Accumulator Register	[no reference]
Stack Pointer	$sp

The stack pointer register follows standard MIPS conventions. Its location can be referenced with "$sp". The accumulator register is the register used for all accumulator logic operations, and therefore does not require a reference tag. Values in the accumulator are not preserved, and it is expected that the stored value in the accumulator will be overwritten after each procedure call. The assembly uses the stack pointer to find which register (or memory address) and take the values from them.

To translate to machine language, you need two sets of data, the opcode for the instruction, and the address that you are addressing with the instruction. Once these are received, you need to translate the hex or decimal values into a binary value. The binary value will be a 16-bit value, where the first 4-bits are the opcode, and the remaining bits are used for the immediates, or addresses.

Euclid’s Algorithm

```
# Euclid's Algorithm
# Written in C by Dr. Sid Stamm and adapted to our assembly
# language by team 02-3T

li    0x9D8
addi  0x9D8
sw    N

sbr   -4
lw    N
sw    1($sp)
jal   relPrime
addi  0           # NOOP
lw    2($sp)
sw    M
sbr   4
j     end         # Terminate the program
addi  0           # NOOP

relPrime:
li    2
sw    M
loopRelPrime:
sbr   -4
lw    N
sw    1($sp)
lw    M
sw    2($sp)
jal   gcd
addi  0           # NOOP
li    1
seq   3($sp)
sbr   4
jnz   endRelPrime      # gcd(n, m) == 1, so finish relPrime
addi  0           # NOOP
lw    M
addi  1
sw    M
```

```
j     loopRelPrime
addi  0           # NOOP

endRelPrime:
lw    M
sw    2($sp)
ret
addi  0           # NOOP

gcd:
li    0
seq   1($sp)
jnz   endGcdB
addi  0           # NOOP

loopGcd:
li    0
seq   2($sp)
jnz   endGcdA
addi  0           # NOOP

lw    1($sp)
sgt   2($sp)
jnz   else
addi  0           # NOOP

lw    2($sp)
sub   1($sp)
sw    2($sp)

j     loopGcd
addi  0           # NOOP

else:
lw    1($sp)
sub   2($sp)
sw    1($sp)
j     loopGcd
addi  0           # NOOP

endGcdA:
lw    1($sp)
sw    3($sp)
```

```
ret
addi  0           # NOOP

endGcdB:
lw    2($sp)
sw    3($sp)
ret
addi  0           # NOOP

end:
lw    M
j     end
addi  0           # NOOP
```

Machine Language Translation of Euclid's Algorithm

*Instructions in blue are no-ops.

Addr.	Translation	Instruction	Jump
0x0800	0011 100111011000	# li 0x9D8	
0x0801	0001 100111011000	# addi 0x9D8	
0x0802	0101 011000000000	# sw N	
0x0803	1110 111111111100	# sbr -4	
0x0804	0100 011000000000	# lw N	
0x0805	0101 100000000001	# sw 1($sp)	
0x0806	1011 000000001101	# jal relPrime	
0x0807	0001 000000000000	# addi 0	
0x0808	0100 100000000010	# lw 2($sp)	
0x0809	0101 011000000001	# sw M	
0x080a	1110 000000000100	# sbr 4	
0x080b	1010 000001000010	# j end	
0x080c	0001 000000000000	# addi 0	
0x080d	0011 000000000010	# li 2	relPrime
0x080e	0101 011000000001	# sw M	
0x080f	1110 111111111100	# sbr -4	loopRelPrime
0x0810	0100 011000000000	# lw N	
0x0811	0101 100000000001	# sw 1($sp)	
0x0812	0100 011000000001	# lw M	
0x0813	0101 100000000010	# sw 2($sp)	
0x0814	1011 000000100100	# jal gcd	

0x0815	0001 000000000000	# addi 0	
0x0816	0011 000000000001	# li 1	
0x0817	0110 100000000011	# seq 3($sp)	
0x0818	1110 000000000100	# sbr 4	
0x0819	1100 000000100000	# jnz endRelPrime	
0x081a	0001 000000000000	# addi 0	
0x081b	0100 011000000001	# lw M	
0x081c	0001 000000000001	# addi 1	
0x081d	0101 011000000001	# sw M	
0x081e	1010 000000001111	# j loopRelPrime	
0x081f	0001 000000000000	# addi 0	
0x0820	0100 011000000001	# lw M	endRelPrime
0x0821	0101 100000000010	# sw 2($sp)	
0x0822	1101 000000000000	# ret	
0x0823	0001 000000000000	# addi 0	
0x0824	0011 000000000000	# li 0	gcd
0x0825	0110 100000000001	# seq 1($sp)	
0x0826	1100 000000111110	# jnz endGcdB	
0x0827	0001 000000000000	# addi 0	
0x0828	0011 000000000000	# li 0	loopGcd
0x0829	0110 100000000010	# seq 2($sp)	
0x082a	1100 000000111010	# jnz endGcdA	
0x082b	0001 000000000000	# addi 0	
0x082c	0100 100000000001	# lw 1($sp)	

0x082d	1000 100000000010	# sgt 2($sp)	
0x082e	1100 000000110101	# jnz else	
0x082f	0001 000000000000	# addi 0	
0x0830	0100 100000000010	# lw 2($sp)	
0x0831	0010 100000000001	# sub 1($sp)	
0x0832	0101 100000000010	# sw 2($sp)	
0x0833	1010 000000101000	# j loopGcd	
0x0834	0001 000000000000	# addi 0	
0x0835	0100 100000000001	# lw 1($sp)	else
0x0836	0010 100000000010	# sub 2($sp)	
0x0837	0101 100000000001	# sw 1($sp)	
0x0838	1010 000000101000	# j loopGcd	
0x0839	0001 000000000000	# addi 0	
0x083a	0100 100000000001	# lw 1($sp)	endGcdA
0x083b	0101 100000000011	# sw 3($sp)	
0x083c	1101 000000000000	# ret	
0x083d	0001 000000000000	# addi 0	
0x083e	0100 100000000010	# lw 2($sp)	endGcdB
0x083f	0101 100000000011	# sw 3($sp)	
0x0840	1101 000000000000	# ret	
0x0841	0001 000000000000	# addi 0	
0x0842	0100 011000000001	# lw M	end
0x0843	1010 000001000010	# j end	
0x0844	0001 000000000000	# addi 0	

0xffff	????????????	0($sp)	
0xfffe	????????????	1($sp)	
0xfffd	????????????	2($sp)	
0xfffc	????????????	3($sp)	
0xfffb	????????????	4($sp)	
0xfffa	????????????	5($sp)	
0xfff9	????????????	6($sp)	
0xfff8	????????????	7($sp)	
0xfff7	????????????	8($sp)	
0xfff6	????????????	9($sp)	
0xfff5	????????????	10($sp)	
0xfff4	????????????	11($sp)	
0xfff3	????????????	12($sp)	
...	...	...	

RTL Diagrams

addi
newPC = PC + 2 PC = newPC inst = Mem[PC] result = acc + SE(inst[11-0]) acc = result

add
newPC = PC + 2 PC = newPC inst = Mem[PC] if(inst[11] == 0) result = ZE(inst[10-0]) else result = Reg[$sp] + inst[10-0] memOut = mem[result] acc = acc + memOut

sub
newPC = PC + 2 PC = newPC inst = Mem[PC] if(inst[11] == 0) result = ZE(inst[10-0]) else result = Reg[$sp] + inst[10-0] memOut = mem[result] acc = acc - memOut

seq

```
                newPC = PC + 2
                  PC = newPC
                inst = Mem[PC]
if(inst[11] == 0) result = ZE(inst[10-0])
   else result = Reg[$sp] + inst[10-0]
          memOut = Mem[result]
       if(acc == MemOut) acc = 1
```

sne

```
                newPC = PC + 2
                  PC = newPC
                inst = Mem[PC]
if(inst[11] == 0) result = ZE(inst[10-0])
   else result = Reg[$sp] + inst[10-0]
          memOut = Mem[result]
       if(acc != MemOut) acc = 1
```

sgt

```
                newPC = PC + 2
                  PC = newPC
                inst = Mem[PC]
if(inst[11] == 0) result = ZE(inst[10-0])
   else result = Reg[$sp] + inst[10-0]
          memOut = Mem[result]
       if(acc > MemOut) acc = 1
```

slt

newPC = PC + 2
PC = newPC
inst = Mem[PC]
if(inst[11] == 0) result = ZE(inst[10-0])
else result = Reg[$sp] + inst[10-0]
memOut = Mem[result]
if(acc < MemOut) acc = 1

j
newPC = PC + 2 PC = newPC inst = Mem[PC] PC = inst[11-0] + 0x0800

jal
newPC = PC + 2 PC = newPC inst = Mem[PC] Mem[$sp] = newPC PC = inst[11-0] + 0x0800

jnz
newPC = PC + 2 PC = newPC inst = Mem[PC] if(acc != 0) PC = inst[11-0] + 0x0800

ret
newPC = PC + 2 PC = newPC inst = Mem[PC] result = Reg[$sp] memOut = Mem[result] PC = memOut

li
newPC = PC + 2 PC = newPC inst = Mem[PC] acc = inst[11-0]

lw
newPC = PC + 2 PC = newPC inst = Mem[PC] if(inst[11] == 0) arg = ZE(inst[10-0]) else arg = Reg[$sp] + inst[10-0] memOut = mem[arg] acc = memOut

sw
newPC = PC + 2 PC = newPC inst = Mem[PC] if(inst[11] == 0) arg = ZE(inst[10-0]) else arg = Reg[$sp] + inst[10-0] Mem[arg] = acc

sbr
newPC = PC + 2 PC = newPC inst = Mem[PC] b = Reg[sp] result = SE(inst[11-0]) b = b + result

RTL Test Procedure

Our RTL Test Procedure substituted arbitrary values in for variables that were unknown, and then traced through the RTL segments in order to determine if the actual value matched the expected value. If it matched, no action was necessary but if not, then the RTL must be adjusted to match the expected value.

RTL Examples

Demonstration of an **`addi`** instruction. This case adds the value in the instruction's bits [11-0] to the accumulator.

```
                              newPC = PC + 2
                                PC = newPC
                              inst = Mem[PC]
                     result = acc + SE(inst[11-0])
                               acc = result
Init Vars:
Acc = 0x915
PC = 0x69

Prgm:
newPC = 0x0069 + 2          → newPC    =    0x006B #newPC=PC+2
PC = 0x006B                 → PC       =    0x006B #PC = newPC
inst = Mem[PC]              → inst     =    0x0420 #Inst
result = 0x915 + 0x420      → result   =    0x0D35 #R = A +
SE[I[11-0]]
acc = 0x0D35                → acc      =    0x0D35 # acc = result
```

Demonstration of an **`add`** instruction. This case adds the value at `0($sp)` (probably a procedure argument) to the accumulator.

```
                              newPC = PC + 2
                                PC = newPC
                              inst = Mem[PC]
                 if(inst[11] == 0) result = ZE(inst[10-0])
                    else result = Reg[$sp] + inst[10-0]
                            memOut = mem[result]
                             acc = acc + memOut
Init Vars:
Acc = 0x915, PC = 0x69, Reg[$sp] = 0x350, mem[0x0770] = 0x963

Prgm:
newPC = 0x0069 + 2          → newPC    =    0x006B #newPC=PC+2
PC = 0x006B                 → PC       =    0x006B #PC = newPC
inst = Mem[PC]              → inst     =    0x0420 #Inst
if(0x0 == 0), R = 0x420;   else R = 0x350 + 0x420
                            → result   =    0x420  #Result ZE(I[10-0])
```

```
memOut = 0x963              → memOut   =    0x0963 # memOut =
M[result]
acc = 0x915 + 0x0963        → acc      =    0x1278 # acc = acc +
memOut
```

Demonstration of an **`sub`** instruction. This case subtracts the value at `0($sp)` (probably a procedure argument) to the accumulator.

```
                        newPC = PC + 2
                          PC = newPC
                        inst = Mem[PC]
            if(inst[11] == 0) result = ZE(inst[10-0])
               else result = Reg[$sp] + inst[10-0]
                       memOut = mem[result]
                        acc = acc - memOut

Init Vars:
acc = 0x925, PC = 0x69, mem[0x0420] = 0x920
Prgm:
newPC = 0x0069 + 2          → newPC    =    0x006B #newPC=PC+2
PC = 0x006B                 → PC       =    0x006B #PC = newPC
inst = Mem[PC]              → inst     =    0x0420 #Inst
(0 == 0)?                   → result   =    0x0420# R=Z(I[10-0])
                                            # else R = R[$sp]+I[10-0]
memOut = mem[0x0420]        → memOut   =    0x0920 #memOut = mem[R]
acc = 0x925 - 0x920         → acc      =    0x005  # acc = acc - memOut
```

Demonstration of an **`seq`** instruction. This case sets the accumulator to a 1 or a 0 depending on whether or not the two arguments are equal or not. (Equal: 1, Not Equal: 0)

```
                        newPC = PC + 2
                          PC = newPC
                        inst = Mem[PC]
            if(inst[11] == 0) result = ZE(inst[10-0])
               else result = Reg[$sp] + inst[10-0]
                       memOut = Mem[result]
                    if(acc == MemOut) acc = 1

Init Vars:
acc = 0x0925
PC = 0x0069
Reg[$sp] = 0x0100
mem[0x0420] = 0x920

Prgm:
newPC = 0x0069 + 2          → newPC    =    0x006B #newPC=PC+2
```

```
inst = Mem[PC]              → inst     =     0x0420 #Inst
if(inst[11] == 0) result = ZE(inst[10-0])
else result = Reg[$sp] + inst[10-0] → result = 0x0200
memOut = Mem[result]        → memOut   =     0x0920
if(acc == MemOut) acc = 1   → acc      =      0x0001
```

This demonstrates the **J** instruction. The program will jump to any address specified to J, as long as it is in range.

```
                          newPC = PC + 2
                            PC = newPC
                          inst = Mem[PC]
                         PC = inst[11-0]

Init Vars:
PC = 0x0069
Inst[11-0] = 0x0200

Pgrm:
newPC = 0x0069 + 2          → newPC    =     0x006B #newPC=PC+2
inst = Mem[PC]              → inst     =     0x0420 #Inst
PC = Inst[11-0]             → PC       =     0x0200 #PC = Inst = 0x0200
```

This demonstrates the **sne** instruction. The program will set acc to 1 or 0 depending if the acc is equal to a specified value at an argument section of the stack.

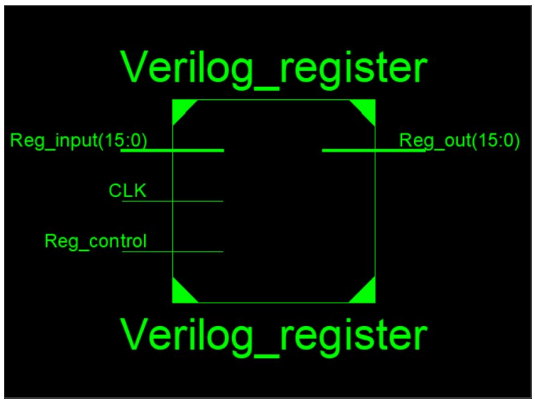
```
                          newPC = PC + 2
                            PC = newPC
                          inst = Mem[PC]
              if(inst[11] == 0) result = ZE(inst[10-0])
                 else result = Reg[$sp] + inst[10-0]
                        memOut = Mem[result]
                     if(acc != MemOut) acc = 1
Init Vars:
PC = 0x0069
acc = 0x925
Reg[$sp] = 0x0100
mem[0x0420] = 0x920

Pgrm:
newPC = 0x0069 + 2          → newPC    =     0x006B #newPC=PC+2
inst = Mem[PC]              → inst     =     0x0420 #Inst
if(inst[11] == 0) result = ZE(inst[10-0]) → result = 0x0420
else result = Reg[$sp] + inst[10-0]
memOut = Mem[result]        → memOut   =     0x920
```

```
if(acc != MemOut) acc = 1  → acc      =     0x0001
```

Components for the Datapath

Components	Input	Output	Control	Description
Register	Reg-in: 16 bits	Reg-out: 16 bits	Reg_control: 1-bit Controls read or write	Accumulator register meant for generic operations and a Stack Pointer Register. It is the block that is called Verilog_Reg that has Reg_in,Reg_out,CLK, and Reg_control
Memory Block	Mem-in: 16 bits	Mem-out: 16 bits	Mem_control: 1-bit Controls read or write	Holds values that we need to save throughout the program.
Gen Op ALU	Instruction: 12 bits Reg-out: 16 bits	ALUA-out - 12 bits	OP-4 bits Determines what operations will occur	Used to calculate arithmetic operations, as well as basic operations.
Extensions	Gen-Op: 11 bits	Mem-input: 16 bits	n/a	Used to sign extend immediates
Mux	Sign extend: 16 bits Zero extend: 16 bits	Mem-input: 16 bits	Extend on: 1 bit	Chooses whether or not we need the sign or zero extend, and this choice is chosen by the extend bit
Comparator	A: 16 bits B: 16 bits	R: 1 bit	n/a	Compares values inputted. It is a block with an A and B input, with an output R.
Address Scaler	A: 16 bits	R: 10 bits	n/a	Scales input memory locations from 16-bit to 10-bit memory space.

Component Specifications

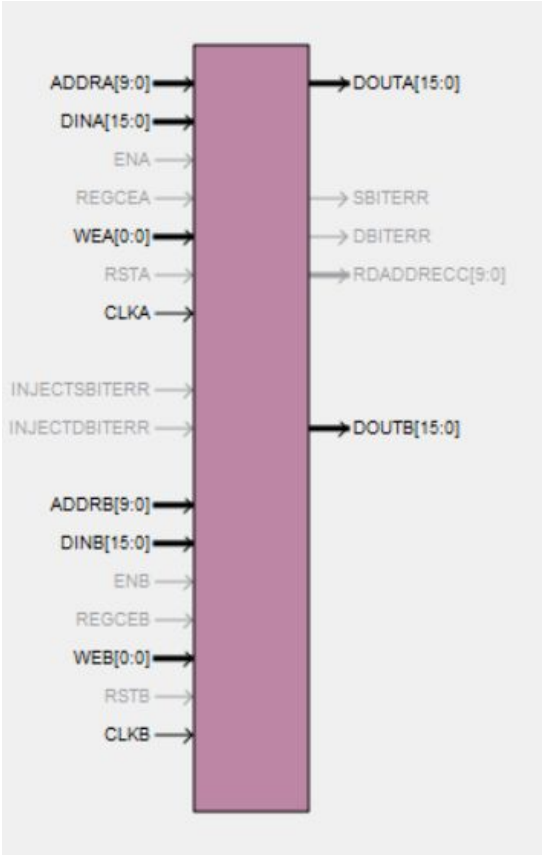


Component: Register

Function: Holds values during common operations and during basic running of the architecture. Will hold values until told otherwise. When told to move the value, it will push it forward. This is triggered by the positive clock edge.

Control Signals: The main control signal is used to either hold a value in the register or push it forward. Reg_control tells Reg_input to either hold its value, or push it to Reg_out.

Testing: The test includes giving the input a value and testing what the value is for the output. The output should not change if control equals zero, and it should change if it equals one.

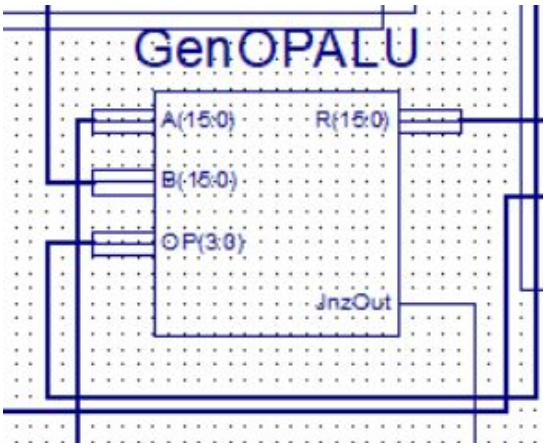


Component: Memory Block

Function: Stores most data used by the processor in the form of 16-bit addressable and writable words. Data includes but is not limited to the complete stack, all instructions, all global variables, input/output values and all OS data.

Control Signals: ADDR is the query address input. DIN is the data in bus (for writing information). WE is the write enable bit. CLK is the clock signal. DOUT is the data out bus. The A/B suffixes of each command indicate which memory access port the signal is attached to.

Testing: Data should be loaded into the memory block in the form of the supplied `memory_small.coe`. The test then checks the first 11 addresses, compares them to the expected values from `memory_small.coe`, and then indicates if the each comparison passed or failed. The test also writes to a single memory location and then reads the written data, and then indicates if the comparison passed or failed.



Component: Gen OP ALU

Function:

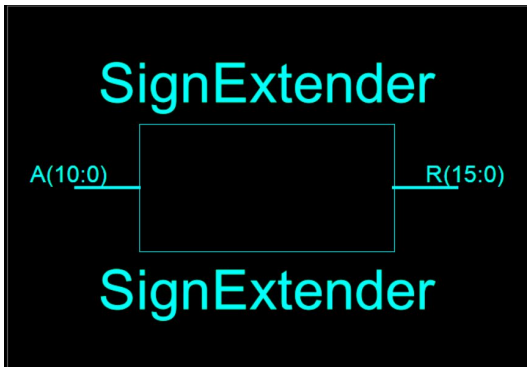
This ALU can perform a few different arithmetic operations. It takes in two input busses A and B which are 16 bits each, and then performs operations using those values, of which the output gets sent to R, a 16 bit output bus. It can add, subtract, compare. or pass through values, dependant on the opcode.

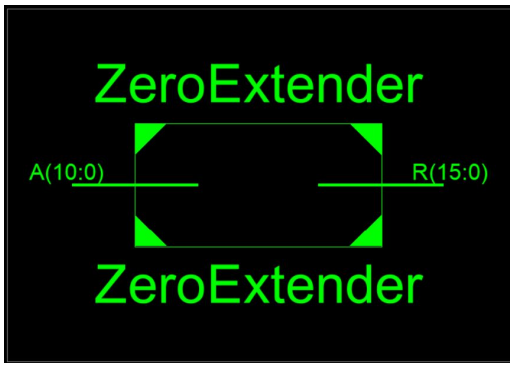
Control Signals:

The control signal for the ALU is the opcode, a four bit value which changes which action the ALU performs on the two 16 bit inputs A and B.

Testing:

The test for the ALU consists of fixing a value of A, and then iterating B from -1 to 3, and the opcode from 0 to 15 for each value of B and then the output was checked against the expected outcome for each arithmetic operation.





Component: Extensions

Function:

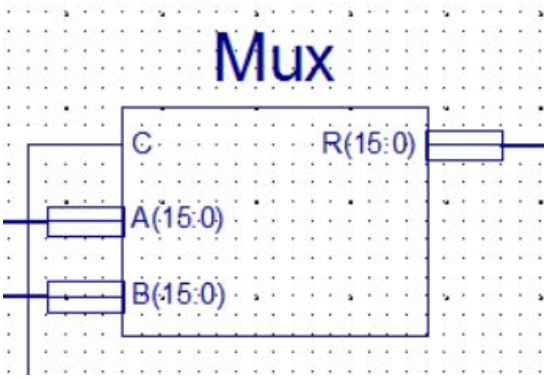
The extenders either do a sign extension or a zero extension to an 11-bit value. The 11-bit value is then extended into a 16-bit value.

Control Signals:

The input is an 11-bit immediate value that is then extended into a 16-bit value, then output.

Testing:

The tests include what the value should look like after the extension and what it will actually be. If the values match, then it is correct. The test also checks if the correct number of bits are output.

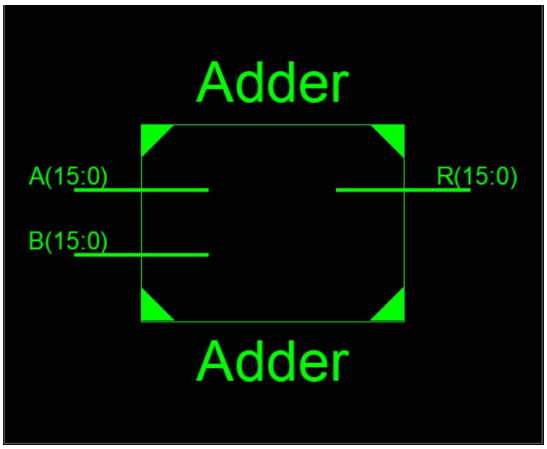


Component: Mux

Function: Takes two 16 bit inputs (A and B). If C = 0, R will equal A, else, R will equal B.

Control Signals: A and B are the 16 bit data inputs that are held. R is the 16 bit data output. There is also a 1 bit control that tells which value moves forward. JnzOut controls the PC source

Testing: The test includes changing the value of C to see which value moves forward. The tests succeeded in determining which value should move forward. The we tested to make sure that when Jnz changed, PC source changed.

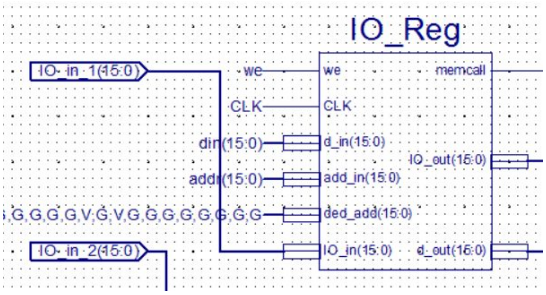


Component: Adder

Function: Takes two 16 bit inputs (A and B). Then, the adder adds the value and outputs the value to R.

Control Signals: A and B are the 16 bit data inputs that are inputted. R is a 16 bit ouput.

Testing: The test includes changing the value of A and B seeing if the R changes appropriately. This change does occur properly and works as expected.

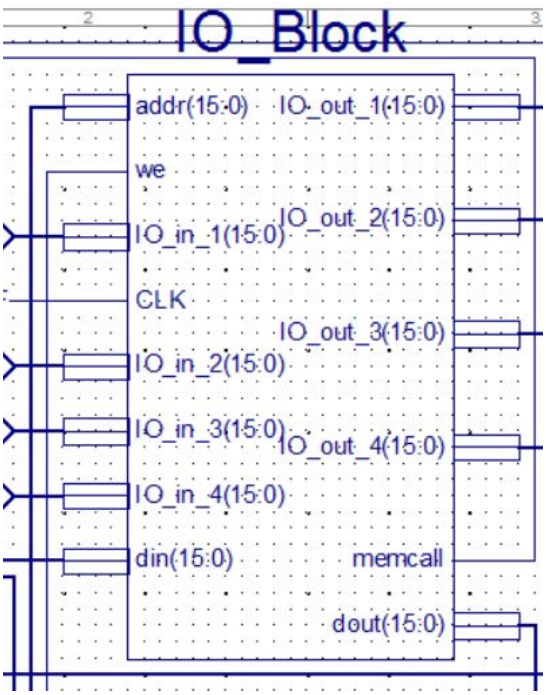


Component: I/O Registers

Function: This register will control the signals and values being sent through I/O. In addition, each register handles one input and one output each. Each register will also have its own corresponding address we write to.

Control Signals: “ded_add” is the inputted address value given to the register. “add_in” is the address gotten from memory. “we” is the write enable gotten from memory. “d_in” is the data retrieved from memory. “IO_in” is the value submitted by I/O. “memcall” is outputted and controls where we get values from, I/O or memory. “IO_out” is the final output if we use I/O. “d_out” is the data that we feed out of the I/O block into the mux that determines where we get data from, the memory block, or I/O.

Testing: To test this Register, we feed values into the we bit and check to see if IO_in works with d_out effectively, and test to see if IO_out interacts with d_in correctly.

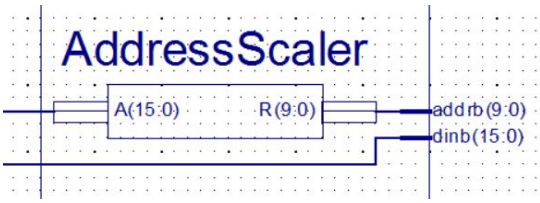


Component: I/O Block

Function: Holds an expandable bank of I/O registers. Intercepts memory busses and outputs data acquired from I/O devices to an output bus, and also outputs to I/O devices from the input bus if writing is enabled. This component functions as an interface between memory and IO ports. The input and output ports can be expanded to cover more memory locations and handle more I/O devices.

Control Signals: “addr”, “we” and “din” function identically to the inputs of a memory block component. If “addr” refers to an I/O location, memcall is set to 1 and “dout” is set to the output value of that I/O register. “IO_in” and “IO_out” are I/O device inputs and outputs that are used to write to and read from I/O registers, so that data can be sent and received appropriately. The number of these ports can be increased if more I/O ports are needed to map I/O memory locations.

Testing: “addr” is sequentially set to the values of each mapped I/O port and “dout” is tested against the I/O input values to test read functionality. Afterwards, “we” is set to 1 and “addr” is cycled again, this time testing the I/O output ports for the value supplied to “din”.

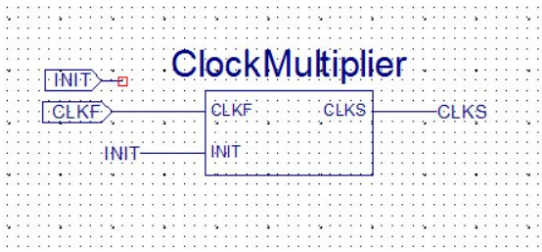


Component: Address Scaler

Function: Translates a single 16-bit address into a 10-bit address that matches our 10-bit memory outline (see the "Memory Allocation" page). This is primarily to match the capabilities of our instruction set. This component can be replaced to scale our instruction set to different memory modules (of different sizes).

Control Signals: A is the 16-bit input signal that will be translated to 10-bit memory space.

Testing: Iterated through the entire memory address space (skipping to every 8th address) and observed the scaler's output to ensure that every address is mapped correctly.



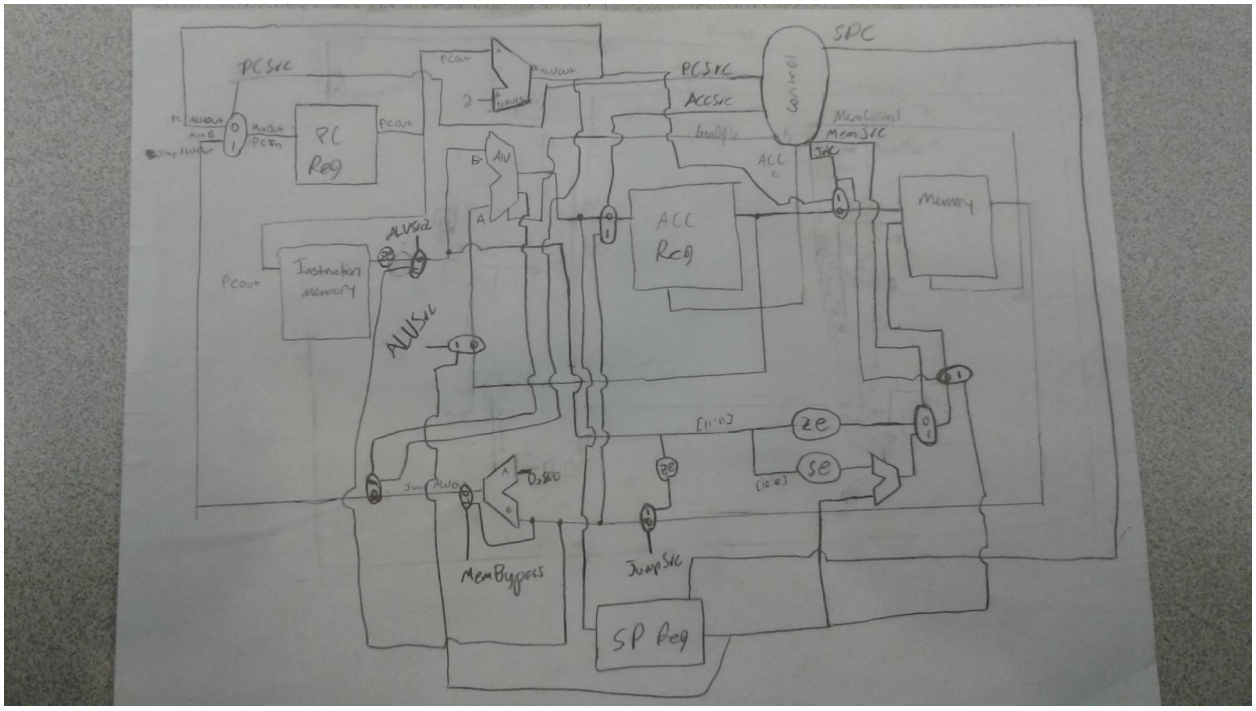
Component: Clock Multiplier

Function: Downscales the input clock signal to be half as fast. This is useful for slowing down the accumulator register update cycle and giving instructions time to execute completely.

Control Signals: INIT is the initialization signal, this sets CLKS to 1 if enabled. CLKF is the source clock input, this clock is downscaled by a factor of 2 and sent to CLKS.

Testing: CLKS is repeatedly cycled. INIT is set to 1 for several clock cycles, then 0 for several clock cycles. While INIT is 1, the output CLKS should be 1. While INIT is 0, CLKS should be a downscaled CLKF signal.

Hand-Drawn Datapath



Integration Plan

After the components are completed and individually tested, the next step involved placing them together in “blocks” to show how they work when implemented in groups. The goal of this is to test whether the individuals work as intended, while also being able to test multiple parts before we implement the full datapath.

Plan:

The plan for implementing the components in groups will be to test components that work well with each other first, to find bugs. We will be connecting an initial set of components, such as the PC register, an adder, and a mux (the PC block), and then add components one at a time until we have a mostly completed datapath. This will allow us to test the functionalities of each of the individual pieces while also testing the capabilities of the components to work with each other.

Test:

We will first combine the PC register, an adder, and a mux to create the PC loop that increments address locations. This will be tested by feeding it values and seeing if it keeps incrementing my the correct value (which is 2).

Next, we will add this PC loop with the instruction memory. The test for this will be to see if the correct instruction is taken out of the instruction memory as the PC loop increments what address we will be at.

After this, we will create the stack pointer register block which contains the stack pointer register, the sign extension and zero extension, and the mux that will determine what value we want. This will connect to the instruction memory. This will be tested by feeding values to the stack pointer register, and testing the zero and sign extension on it. Then, we will test control signals to see if the outcomes are correct.

The next components will be the Accumulator register and the Gen Op ALU. We will test if the values and op that goes into the ALU are correct, and then we will see if the Accumulator will work correctly by holding and pushing values. To do this, we will also need to add the memory block with this as well as the control unit. These will be added all at the same time.

Next, we had to make specific changes to the design of our datapath. These included adding some adders and muxes for specific instructions (see updated datapath to view these additions). The muxes helped identify bugs within the instructions and datapath.

Creating the Control Units

The purpose of the control units are to control what components of our datapath do at any point as the processor cycles through the path.

Signal	Description
Acc_Write	Determines if the accumulator is written to
Acc_Src	Determines the accumulator write input value
PC_Src	Determines if we do a jump or not
SP_Write	Determines if we write to the stack pointer
Mem_Write	Determines if we write to memory
Mem_Src	Determines the source of the memory module's address
Jump_Src	Determines if a jump occurs
JalC	Determines if the Jal command is being used. Saves PC+2
ALU_Src	Determines the 'A' GenOp input value.
ALU_Src2	Determines the 'B' GenOp input value.
MemBypass	Determines if we use ret
GenOp	Determines what operation is done in the ALU

Control Unit Output

[X] = Don't Care, [?] = Instruction Determined

Instruction	Acc Write	Acc Src	PC Src	SP Write	Mem Write	Mem Src	Jal C	Jump Src	ALU Src	ALU Src 2	Mem Bypass	Gen Op
add	1	0	0	0	0	?	0	0	0	1	0	0000
addi	1	0	0	0	0	0	0	0	0	0	0	0001
sub	1	1	0	0	0	?	0	0	0	1	0	0010
li	1	0	0	0	0	0	0	0	0	0	0	0011
lw	1	1	0	0	0	?	0	0	0	0	0	0100
sw	0	X	0	0	1	?	0	0	0	0	0	0101
seq	1	0	0	0	0	?	0	0	0	1	0	0110
sne	1	0	0	0	0	?	0	0	0	1	0	0111
sgt	1	0	0	0	0	?	0	0	0	1	0	1000
slt	1	0	0	0	0	?	0	0	0	1	0	1001
j	0	0	1	0	0	?	0	1	0	0	0	1010
jal	0	0	1	0	1	?	1	1	0	0	0	1011
jnz	0	0	1	0	0	?	0	1	0	0	0	1100

ret	0	0	1	0	0	?	1	0	0	0	1	1101
sbr	0	0	0	1	0	?	0	0	1	0	0	1110

Testing the Control Unit

The control unit takes an input that is a 16 bit instruction, and depending on the instruction, different control outputs will occur. To test the control unit, we must first identify what 16 bit value corresponds with each instruction. For the tests themselves, we look at the first 4 bits, which indicate the Opcode, and the 1 bit after the Opcode, which indicates the special bit. The test will run through every Opcode, and then it will test each control signal value to see if they match what we need for the instruction. To complete this, we will also need to look at the special bit. If the special bit is 1, then the control signal Mem Src will also be 1. If it is zero, the control signal be zero. If the control unit works correctly, the control unit will match all the instructions to their Opcodes, and their control signals.

Testing the Full Implementation of the Datapath

Once the individual tests are complete, we will start the test of the fully implemented datapath. The steps for testing the full implementation is simple. First, we check all wires and signals to see if the setup is correct. Then, we run some test programs that will run individual tests on specific instructions. Once the program passes those tests, we will then go on to the euclid's algorithm that will be used to test our processor in its entirety. Some of the tests for the datapath is written below.

```
# Test code for datapath
# This program tests arithmetic operations and memory
# to make sure they work properly.
# features: add, addi, sub, li, lw, sw
# At program end:
# A: M(0x40) = 24; B: M(0x42) = 34; C: M(0x44) = -18;

li 8        # Loads 8 into accumulator
addi 16     # 8 + 16 = 24
sw A        # A = 24

li 10       # Loads 10 into acc
sw B        # B = 10

lw B        # loads B = 10
add A       # performs 24 + 10
sw B        # B = 34

li 6        # loads 6 into acc
sw C        # C = 6

lw C        # Loads C = 6
sub A       # Performs 6 - 24
sw C        # C = -18
```

Addition Test

This Test allows us to see how arithmetic operations will work with memory, as well as being able to see if the ALU is working as expected. This test also helped debug that specific block of the datapath.

```
# Test code for datapath
# This program tests jump

li 4
sw A        # A = 4
j label

li 7        # Happens if j label doesn't work
sw B        # B = 7

label:

li 6        # C = 6
sw B

j end

li 5        # D = 5
sw C        # Only happens if j end doesn't work

end:
```

Jump Test

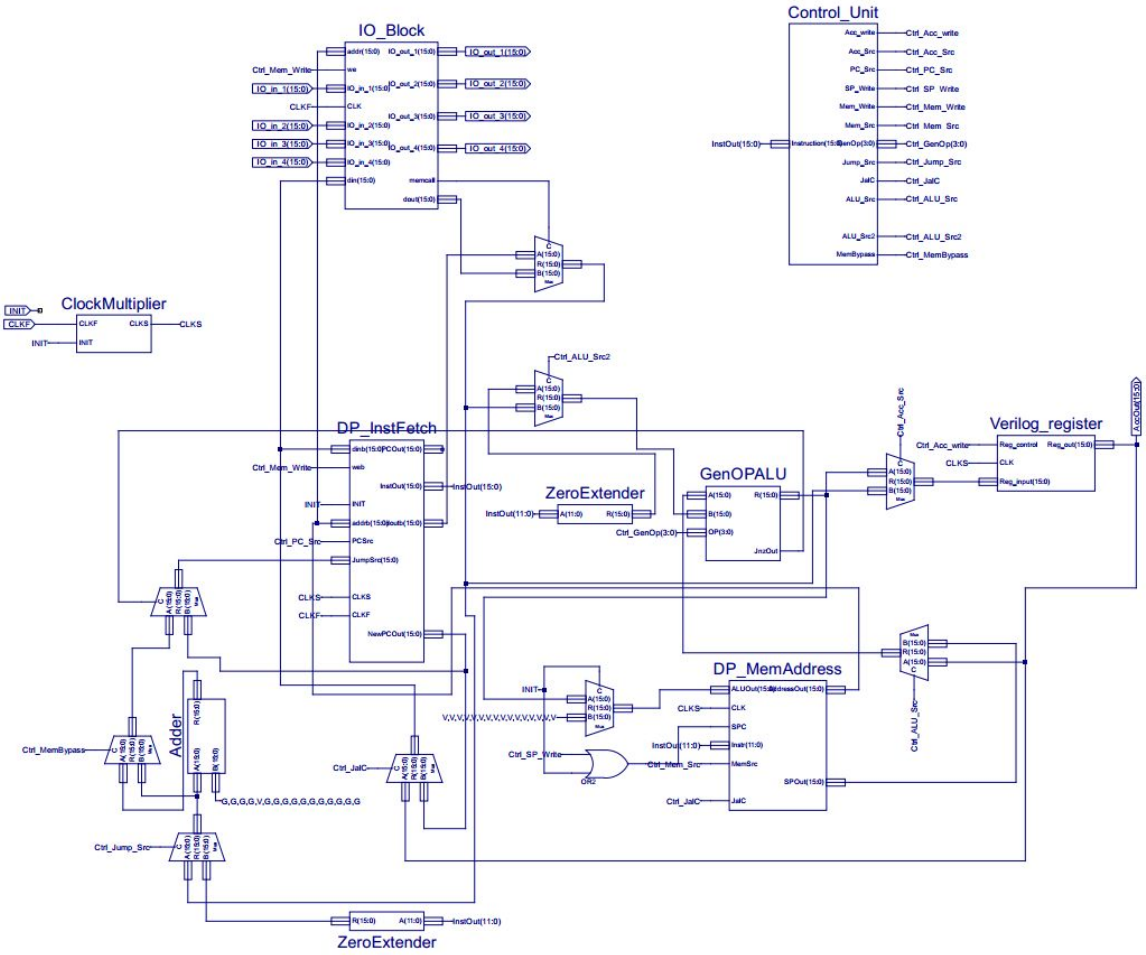
This Test allows us to see how the jump instruction functions. Since the jumps are a vital portion of the instructions, this test helped debug problems within the jump commands .

```
# Test code for datapath
# This program tests return
# At prgm end, A = 8 is working properly

li 2
sw A        # A = 2

jal label   # goto label
addi 0      # nop

li 4
add A
sw A        # A = A + 4

label:
    li 4
    sw A    # A = 4
    ret
```

Return/Jal Test

This Test allows us to see how the return instruction as well as the Jump and link instruction work together. This test allowed us to find bugs within the return and jal instruction, which helped complete the datapath.

Finished Datapath



This is the finalized datapath that completes Euclid's algorithm with the relprime function included. Because of many tests being implemented throughout different times of datapath construction, many pieces were added after some blocks were formed, like the multiplexers and the adders seen above.

Benchmarking

Bytes: 69 instructions + 2 global variables = 71 words = 142 bytes
Total Instructions: 132727
Number of Cycles: 132727
Cycles per Instruction: 1
Clock Speed: 50.26 mHz

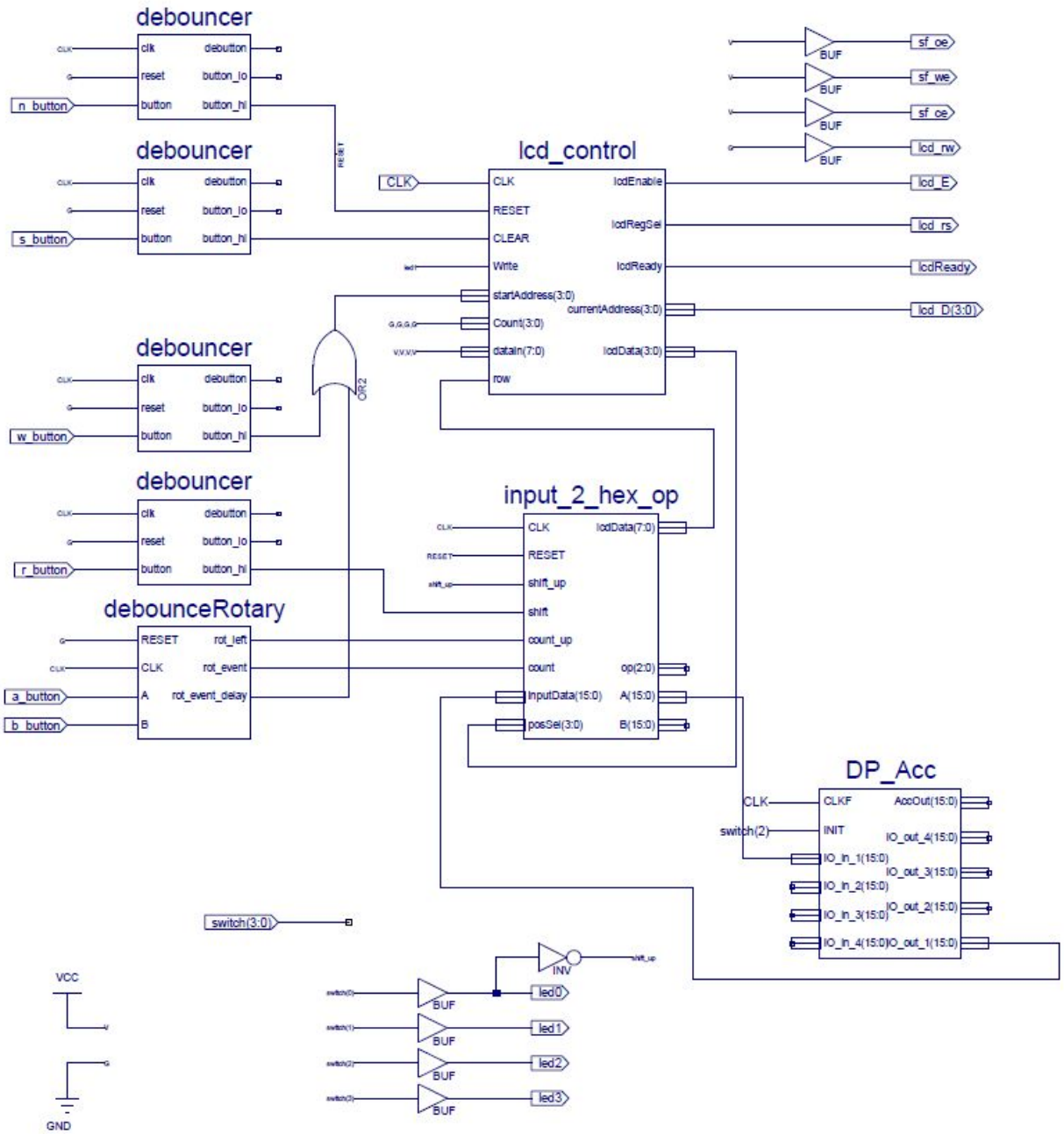
Device utilization summary:

Selected Device : 3s500efg320-4

Number of Slices:	311	out of	4656	6%	
Number of Slice Flip Flops:	202	out of	9312	2%	
Number of 4 input LUTs:	585	out of	9312	6%	
Number of IOs:	146				
Number of bonded IOBs:	146	out of	232	62%	
Number of GCLKs:	1	out of	24	4%	

FPGA Implementation

FPGA_IO Main datapath:



To integrate our processor with the FPGA board, we took the format of ALUIO (Provided by Sid) and modified it in order to fit our needs. Our processor that we made was called DP_Acc and was used as the output. The input_2_hex_op was modified in order to allow the LCD to display the correct values. The main modifications were to clear the center of the LCD so only the value and what the relatively prime value was. We decided to use switch(2) to let our processor know when to start calculations.