Aristotle

Qingyuan Jiao

JL Koenig

Ryan Bowering

Canon Maranda

Table of Contents

Description

Our architecture is derived from the accumulator architecture, designed to be cost-effective and have a simpler instruction set than popular alternatives by limiting the number of registers available. Our architecture uses one 16-bit common register, 16-bit words and 16-bit addresses. In addition, there are two special purpose registers: the stack pointer and the program counter.

Registers Available

Register	Name	Use	Saver
x0	cr	Common use register	Caller
x1	sp	Stack pointer	Callee
x2	pc	Program counter	-

I/O

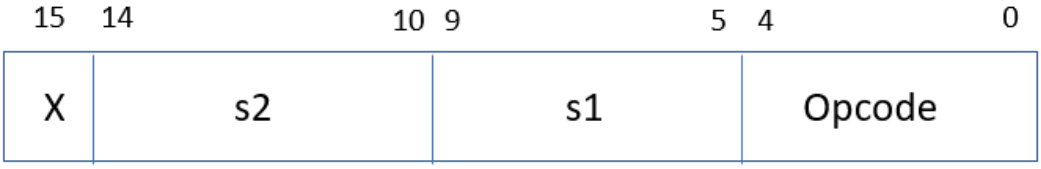
I/O of RelPrime is implemented through two words above the stack frame of RelPrime function. Input into RelPrime is placed on word 255, which is the location allocated for the first parameter of RelPrime. The output will be stored at word 254, which is the location designed for RelPrime’s return value.

Instructions

Type C

C (constant) type instructions contain a 5-bit Opcode and a 11-bit signed constant value.

Type S



S (standard) type instructions contain a 5-bit Opcode, two 5-bit constants s1 and s2 (offset 1 and offset 2) used to access 2 different locations on stack offset from sp by s1 and s2 respectively, and one bit X that is not used for anything. s1 and s2 are unsigned.

As far as conversion between aristotle and machine code:

- δ Check the type of instruction and reference guidelines above
- δ For type S, assume s2 and s1 are unsigned and used relative to SP to determine the addresses to get values from. Their range is SP to SP+31.
- δ For branch instructions, bpc will apply a constant offset to current pc. Other branching instructions will set PC to the current value in cr register.

Mnemonic	Type	Name	Opcode	Description
add	S	ADD	00010	R[cr] = R[cr] + M[sp+s1]
addc	C	ADD Const	00001	R[cr] = R[cr] + const
addcsp	C	ADD Const Stack Pointer	00101	SP = SP + const
and	S	AND	10110	R[cr] = R[cr] & M[sp+s1]
andc	C	AND Const	10101	R[cr] = R[cr] & const
or	S	OR	01110	R[cr] = R[cr] | M[sp+s1]
orc	C	OR Constant	10001	R[cr] = R[cr] | const
sub	S	SUBtraction	00110	R[cr] = R[sp+s1] - M[cr]
sl	C	Shift Left	01001	R[cr] << const
sr	C	Shift Right	01101	R[cr] >> const
lds	S	LoaD from Stack	11010	R[cr] = M[sp+s1]
wrs	S	WRite to Stack	10010	M[sp+s1] = R[cr]
ssp	C	Set Stack Pointer	00011	SP = R[cr]
lpc	S	LoaD PC	00111	R[cr] = PC
lsp	S	LoaD SP	01011	R[cr] = SP
lu	C	Load Upper bits	10100	R[cr] = {const, 6b'0}
b	C	Branch	11100	PC = R[cr]
beq	S	Branch EQual	00000	if (M[sp+s1] == M[sp+s2]) PC = R[cr]
bge	S	Branch Greater than or Equal	00100	if (M[sp+s1] >= M[sp+s2]) PC = R[cr]
bgt	S	Branch Greater Than	01000	if (M[sp+s1] > M[sp+s2]) PC = R[cr]
bneq	S	Branch Not EQual	01100	if (M[sp+s1] != M[sp+s2]) PC = R[cr]
bpc	C	Branch relative to PC	11111	PC = PC + const
end	C	END of the program	11011	This instruction marks the end of execution

Pseudo instruction	Name	Description
addow	ADD OverWrite	R[cr] = M[sp+s1] + M[sp+s2]
call	Call branch	R[cr] = PC + 6 M[SP] = R[cr] PC = PC + s1
ld	Load label	R[cr] = PC + s1

Instruction Class	**Instruction Name**	**Aristotle Key**	**Function**
ALU Commands	Add	add	Adds a value from memory to cr
	Add Constant	addc	Adds a constant to cr
	Add Constant SP	addcsp	Adds a constant to SP
	Add Overwrite	addow	Adds two values from memory and stores them in cr
	And	and	Bitwise AND operation between cr and value from memory
	And Constant	andc	Bitwise AND operation between cr and constant value
	Or	or	Bitwise OR operation between cr and value from memory
	Or Constant	orc	Bitwise OR operation between cr and constant value
	Subtract	sub	Subtracts a cr from a value in memory
	Shift Left	sl	Shifts the value in cr left const bits
	Shift Right	sr	Shifts the value in cr right const bits
Read Write Commands	Load from Stack	lds	Loads a value from the stack into cr
	Write to Stack	wrs	Writes the value from cr onto the stack
	Set Stack Pointer	ssp	SP becomes the value in cr
	Store from address to stack	sas	Reads the value in memory addressed by cr and stores it somewhere on the stack relative to SP
	Store from stack to address	ssa	Does the opposite of sas. Takes a value stored somewhere on the stack relative to PC and put the value to the memory space addressed by cr.
	Load PC	lpc	Loads the current value of PC into cr
	Load SP	lsp	Loads the current value of SP into cr
	Load Upper bits	lu	Loads a constant as the most significant 10 bits into a register and places 0's as the lower 6
Branch Commands	Branch	b	Moves PC to value in cr
	Branch Equal	beq	Moves PC to value in cr if two places on the stack store equal values
	Branch Not Equal	bne	Moves PC to value in cr if two places on the stack do not store equal values

	Branch Greater Than	bgt	Moves PC to value in cr if value at s1 > value at s2 (where s1 and s2 are relative positions on the stack)
	Branch Greater Than or Equal	bge	Moves PC to value in cr if value at s1 >= value at s2 (where s1 and s2 are relative positions on the stack)
	Branch Relative to PC	bpc	Moves PC the number of instructions given by a constant

Calling Conventions

Caller requirements:

1. Decrement sp by 1 + the number of arguments passed to the callee (If callee has no arguments, decrement sp by 1 to reserve a location for callee's return value)
2. Store any arguments directly above M[sp] on stack (starting at M[sp + 1])
3. Load pc into cr, add 3 to it, and store it to M[sp]
4. Load address of callee to cr
5. Branch to callee

Callee requirements:

1. At the end of execution, store return value in cr register
2. Reset sp so that it points to callee's return address
3. Store return value to the first word above the return address (Mem[sp+1]. i.e., where the first argument is stored)
4. Load return address (M[sp]) into cr
5. Branch back to caller

Stack Diagram

Note: The arg... can be expanded for as many arguments as the function requires

<table>
<tr><td>Caller Local</td><td></td><td rowspan="5">SP During Caller</td></tr>
<tr><td rowspan="4">Caller Setup for Callee</td><td>arg...</td></tr>
<tr><td>arg1</td></tr>
<tr><td>arg0</td></tr>
<tr><td>Return Address</td></tr>
<tr><td>Callee Local</td><td></td><td>SP During Callee</td></tr>
</table>

Assembly Translation of the Program

Assembly Instructions		Machine Code
RelPrime:		RelPrime:
addcsp -2		11111111110 00101
andc 0		00000000000 10101
addc 2		00000000010 00001
wrs 0	//m=2	X XXXXX 00000 10010
addcsp -3		11111111101 00101
Loop: (5)		Loop:
lds 6	//cr=n	X XXXXX 00110 11010
wrs 1	//prepare n	X XXXXX 00001 10010
lds 3		X XXXXX 00011 11010
wrs 2	//prepare m	X XXXXX 00010 10010
call gcd	//+3	X XXXXX XXXXX 00111
		00000000011 00001
		X XXXXX 00000 10010
//bpc 14		00000001110 11111
andc 0		00000000000 10101
addc 1		00000000001 00001
wrs 4	//putting 1 on stack	X XXXXX 00100 10010
ld RPRet	//+6	X XXXXX XXXXX 00111
		00000000110 00001
beq 1, 4	//compares return value to 1	X 00100 00001 00000
lds 3		X XXXXX 00011 11010
addc 1		00000000001 00001
wrs 3		X XXXXX 00011 10010
bpc Loop	//-18	11111101110 11111
RPRet: (23)		RPRet:
lds 3		X XXXXX 00011 11010
addcsp 5		00000000101 00101
wrs 1	//stores m as return value	X XXXXX 00001 10010
lds 0	//gets the return address	X XXXXX 00000 11010
b		XXXXXXXXXXX 11100
wrs 0		X XXXXX 00000 10010
end		XXXXXXXXXXX 11011
gcd: (27)		gcd:
addcsp -1		11111111111 00101
andc 0		00000000000 10101
wrs 0		X XXXXX 00000 10010
ld while	//pseudo instruction	X XXXXX XXXXX 00111
	//+5	00000000101 00001
bneq 0, 2		X 00010 00000 01100
lds 3		X XXXXX 00011 11010

```
      wrs 2                                          X XXXXX 00010 10010
      bpc exit    //+14                              00000001110 11111
while: (36)                                    while:
      ld exit     //pseudo instruction               X XXXXX XXXXX 00111
                  //+13                              00000001101 00001
      beq 3, 0                                       X 00000 00011 00000
      ld else                                        X XXXXX XXXXX 00111
                  //+6                               00000000110 00001
      bge 3, 2                                       X 00010 00011 00100
      lds 3       //cr=b                             X XXXXX 00011 11010
      sub 2       //cr=a-b                           X XXXXX 00010 00110
      wrs 2       //a=cr                             X XXXXX 00010 10010
      bpc while   //-10                              11111110110 11111
else: (46)                                     else:
      lds 2       //load a                           X XXXXX 00010 11010
      sub 3       //cr=b-a                           X XXXXX 00011 00110
      wrs 3       //b=cr                             X XXXXX 00011 10010
      bpc while   //-14                              11111110010 11111
exit: (50)                                     exit:
      addcsp 1                                       00000000001 00101
      lds 0                                          X XXXXX 00000 11010
      b                                              XXXXXXXXXXX 11100
```

In a normal execution, instructions in red will be executed while instructions in green will not. For testing purposes, however, green instructions will be executed instead in the testbench.

Pseudo Instruction Translation

```
call gcd:
      lpc
      addc 3      //calculate the ra for gcd
      wrs 0       //put the return address on stack
      bpc gcd     //PC-offset of label gcd: 18(*2)

ld RPRet:
      lpc
      addc 6      //loads the address of the RPRet

ld while:
      lpc
      addc 5      //set cr to address of while

ld exit:
      lpc
      addc 13     //exit address

ld else:
      lpc
      addc 6
```

Assembly Language Fragments

Where s1, s2 are 5-bit values and const is a 10-bit value

Assembly Instructions	**Machine Code**
Reading Data:	
lds 5 // Loads sp + 2*(5) into cr	X XXXXX 00101 11010
Writing Data:	
wrs 4 // Stores cr into sp + 2*(4)	X XXXXX 00100 10010
Loading Address into Register:	
lsp // Stores current stack pointer into cr	X XXXXX XXXXX 01011
Iteration:	
Label:	
lds 8 // Loads sp + 2*(8) into cr	X XXXXX 01000 11010
addc 1 // Sets cr to cr + 1	00000000001 00001
wrs 8 // Stores cr into sp + 2*(8)	X XXXXX 01000 10010
bpc label //branches to label	11111111100 11111
Multiplication:	
lds 6 // Loads sp + 2(6) into cr	X XXXXX 00110 11010
sl 2 // Sets cr to cr * 2^2	00000000010 01001
wrs 6 // Stores cr into sp + 2(6)	X XXXXX 00110 10010
Conditional Statements:	
lpc // Stores current program counter into cr	X XXXXX XXXXX 00111
addc 6 // Increments the value inside cr by const	00000000110 00001
beq 1, 0 // Branches to cr if Mem[SP+2*(1)] == Mem[SP]	X 00000 00001 00000
Preparing Function Arguments:	
addcsp -2 // Decrement sp by 2*(2)	11111111100 00101
wrs 1 // Stores cr into sp + 2*(1)	X XXXXX 00001 10010
lpc // Stores current program counter into cr	X XXXXX XXXXX 00111
addc 6 // Adds 6 to cr	00000000110 00001
wrs 0 // Stores cr into sp	X XXXXX 00000 10010
bpc function // Branches to function (PC-relative)	IAmAddress 11111
(The last three function can be replaced by pseudo instruction “call function”)	
Returning:	
addcsp 8 // Moves sp to the top of callee’s stack frame	00000000100 00101
wrs 1 // Stores cr(containing return value) into sp + 2	X XXXXX 00001 10010
lds 0 // Sets cr to sp	X XXXXX 00000 11010
b // Branches to cr	X XXXXX XXXXX 11100

RTL Instructions

Branching Instructions

	Conditional Branching (Literal)	**PC-Relative Branching**
Instruction Fetch & Decode	PC <= PC + 1 IR <= Mem[PC]	
Execute & Early Writeback	ALU1 <= SP + imm(IR[9:5]) ALU2 <= SP + imm(IR[14:10])	PC <= PC + imm(IR[15:5])
Memory	Mem1 <= Mem[ALU1] Mem2 <= Mem[ALU2]	
Writeback	if (Mem1 == Mem2)* PC <= CR	

**Can be replaced with >=, >, !=, or no condition*

ALU(Computational) Instructions

	S Type	**C Type**
Instruction Fetch & Decode	PC <= PC + 1 IR <= Mem[PC]	
Execute & Early Writeback	ALU1 <= SP + imm(IR[9:5])	CR <= CR op imm(IR[15:5])*
Memory	Mem1 <= Mem[ALU1]	
Writeback	CR <= CR op Mem1	

*CR can be replaced with SP

Memory(Stack) Instructions

	lds	**wrs**
Instruction Fetch & Decode	PC <= PC + 1 IR <= Mem[PC]	
Execute & Early Writeback	ALU1 <= SP + imm(IR[9:5])	
Memory	CR <= Mem[ALU1]	Mem[ALU1] <= CR
Writeback		

Register-to-Register Instructions

	lpc	**lsp**
Instruction Fetch & Decode	PC <= PC + 1 IR <= Mem[PC]	
Execute & Early Writeback	CR <= PC	CR <= SP
Memory		
Writeback		

Components List

- PC Register (Register.v)
  - One 16-bit input
  - One 16-bit output
  - One 1-bit CLK signal
  - One active-high synchronous reset bit
  - One control bit (WRT) that controls writing to the register. If it's 1, then writing to the register is enabled.
  - Storage for our PC value. On a rising edge of the CLK, if the control bit is 1, output will be set to the input.
  - This will implement the PC symbol in our RTL.
- CR Register (Register.v)
  - One 16-bit input
  - One 16-bit output
  - One 1-bit CLK signal
  - One active-high synchronous reset bit
  - One control bit (WRT) that controls writing to the register. If it's 1, then writing to the register is enabled.
  - Storage for our CR value. On a rising edge of the CLK, if the control bit is 1, output will be set to the input.
  - This will implement the CR symbol in our RTL.
- SP Register (sp_register.v)
  - One 16-bit input
  - One 16-bit output
  - One 1-bit CLK signal
  - One active-high synchronous reset bit
  - One control bit (WRT) that controls writing to the register. If it's 1, then writing to the register is enabled.
  - Storage for our SP value. On a rising edge of the CLK, if the control bit is 1, output will be set to the input.
  - This will implement the SP symbol in our RTL.
- Instruction D Latch (denoted by IR, implemented by D_Latch.v)
  - One 16-bit input
  - One 16-bit output
  - One 1-bit CLK signal
  - One active-high synchronous reset bit
  - One control bit (WRT) that controls writing to the register. If it's 1, then writing to the register is enabled.

- σ Storage for instruction being executed. When CLK is high, output is set to input.
- σ This will implement the IR symbol in our RTL.

δ Memory (DistributedRAM.v) and Mem1, Mem2 registers (Register.v)

- σ Memory is implemented by dual-port distributed memory. It supports asynchronous reading and synchronous writing. It consists of Instruction Memory, Stack Memory. The memory can store 256 words in total (word 0 – word 255), addressed by 8 bits. It has two 8-bit reading ports (Read1, Read2), two 16-bit output ports (Out1, Out2), one 8-bit write address port (Addr), one 16-bit write data port (Value), one control bit (we, writeEnabled) that controls writing into the memory.
- σ Instruction Memory only supports reading. Lower 8-bits of PC is fed into read1, and out1 is fed into IR. we is always disabled. Other ports are not used.
- σ Stack Memory uses both read address inputs (Read1, Read2) and the corresponding data outputs (Out1, Out2) for memory reading. Memory writing ports are also used. we is controlled by memWrite control signal.
- σ Out1 and Out2 will be stored in Registers Mem1 and Mem2. These registers have the same architecture as registers mentioned above and will be used to transfer values from step 3 to step 4 in our RTL.
- σ This will implement the Mem[] symbol in our RTL.

δ ALU1, ALU2 (ALU.v), and ALUO1, ALUO2 registers (Register.v)

- σ Each ALU has two 16-bit inputs (A, B), One 16-bit output (ALUOut), Two 1-bit outputs (zeroDetect, carryout), a 3-bit control signal.
- σ In regards to the 3-bit control signals 000: add, 001: subtract, 010: shift left, 011: shift right, 100: or, 101: and.
- σ Performs Arithmetic operations on A and B based off the control signal, zeroDetect is high when the operation returns 0. Result of the operation returned in the 16-bit output.
- σ Implements A op B, ALUOut, SP + X, PC + X, and all comparison operations (==, >= <=…) in our RTL.
- σ Outputs of ALUs are stored in ALUO1 and ALUO2 registers respectively to transfer values from step 2 to step 3 in RTL. These registers have the same architecture as registers mentioned above.

δ Control unit (ControlUnit.v)

- σ One 5-bit Opcode input, one 1-bit CLK, one 1-bit RST signal, twenty 1-bit control signal outputs

- ω PCSrc – determines if PC is set to the value stored in CR or the operation performed by ALU1
- ω CRSrc – selects out1 of the stack memory, ALU1Out, ALU2Out, or 0
- ω IRWrite – determines if instruction memory will write to the instruction D latch
- ω SPWrite – determines if the output of ALU2 will write to the stack pointer register
- ω MemWrite – determines if the stack memory will be written to
- ω PCWrite – determines if the program counter register will be written to
- ω CRWrite – determine if the common use register will be written to
- ω PCWriteCond – only high if the current instruction is a conditional branching
- ω BType – the condition or type of branch instruction
- ω ALU1Src1 – selects 0, 2, s1, or const
- ω ALU1Src2 – selects PC or SP
- ω ALU2Src1 – selects 0, const, s2, or Mem1
- ω ALU2Src2 – selects 0, SP, CR, or Mem2
- ω ALU1Op – controls the function performed by ALU1
- ω ALU2Op - controls the function performed by ALU2
- ω ALUO1WRT – determines if ALU1Out will be written to ALUO1 register
- ω ALUO2WRT – determines if ALU2Out will be written to ALUO2 register
- ω MemO1WRT – determines if Out1 of the stack memory will be written to Mem1 register
- ω MemO2WRT – determines if Out2 of the stack memory will be written to Mem2 register
- ω EOP – high if the current instruction is end. Marks the end of execution

  - σ This element disperses and decides what values the control signals will be for all our components

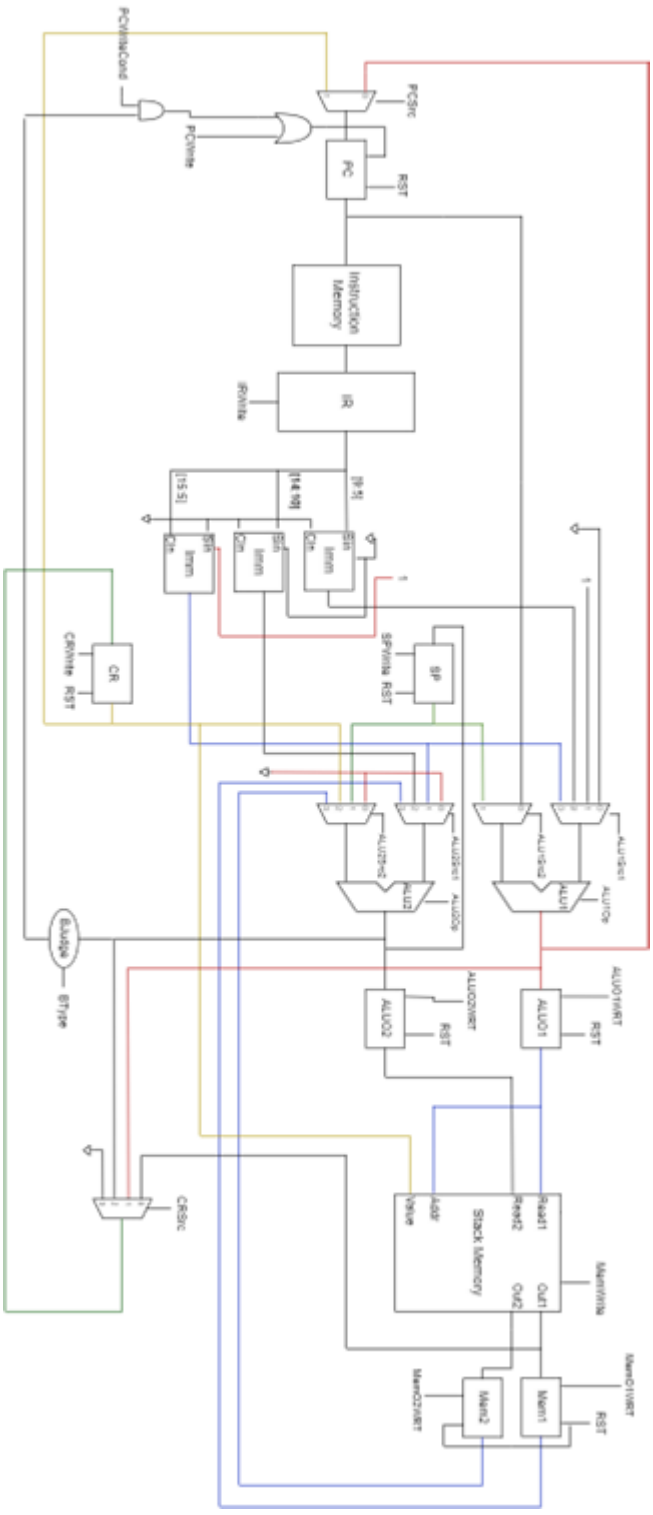
δ Immediate generators (ImmediateGenerator.v)

  - σ One 5-bit input (Sin), one 11-bit input (Cin), one 1-bit control (Select), and one 16-bit output.
  - σ If Select is 0, ImmGen will pad upper 11 bits of Sin with 0 and use it as the output. If Select is 1, ImmGen will sign-extend Cin to 16-bit and use it as the output.
  - σ Implements `imm(IR[15:5])`, `imm(IR[14:10])`, `imm(IR[9:5])` in our RTL

δ Branching Judge (BranchingJudge.v)

- One 16-bit input (in), one 1-bit output (out), one 2-bit control (BType).
- BType corresponds to the branching type of the current instruction. 00: beq, 01: bge, 10: bgt, 11: bneq.
- Case 1: beq. Output is 1 when input is 0.
- Case 2: bge. Output is 1 when input >= 0.
- Case 3: bgt. Output is 1 when input > 0.
- Case 4: bneq. Output is 1 when input is NOT 0.

Datapath Diagram



Test Plans

- Registers (Register_tb.v)
  1. All registers have the same architecture so they will be tested altogether.
  2. Set RST to 1 for one clock cycle. Verify that the output is correctly set to 0. For sp register, the output will be set to 254.
  3. Set input to a non-zero value for one clock cycle. Set the control bit WRT low. Verify that the output stays 0.
  4. Set the WRT high with input unchanged for one clock cycle. Verify that the output is set to the input at the rising edge of CLK.
  5. Set input to a different value with WRT high for one clock cycle. Verify that the output is set to the input at the rising edge of CLK.
  6. Set RST to 1 for one clock cycle. Verify that the output resets to 0.

- Immediate Generators (ImmGen_tb.v)
  1. Set Sin to a random value. Set Cin to a random positive value (first bit is 0). Set Select to 0. Wait for 10ns. Verify that the output is equal to Sin.
  2. Set Select to 0. Wait for 10ns. Verify that the output is equal to Sin.
  3. Set Select to 1. Wait for 10ns. Verify that the output is equal to Cin.

- Branching Judge (BJudge_tb.v)
  1. Set the input to 0.
  2. Set BType to 00 (beq). Wait for 10ns. Verify that the output is 1.
  3. Set BType to 01 (bge). Wait for 10ns. Verify that the output is 1.
  4. Set BType to 10 (bgt). Wait for 10ns. Verify that the output is 0.
  5. Set BType to 11 (bneq). Wait for 10ns. Verify that the output is 0.
  6. Set the input to a positive value.
  7. Repeat step 2-5. This time, verify that the values are 0, 1, 1, 1, respectively.
  8. Set the input to a negative value.
  9. Repeat step 2-5. This time verify that the values are 0, 0, 0, 1, respectively.

δ Stack Memory (DistributedRAM_tb.v)
SP can be any value between [0-254]
1. Set Value to 5. Set Addr to SP. Wait for one clock cycle.
2. Set Read1 to SP. Verify that Out1 changes to 5 instantly.
3. Set Value to 7. Set Addr to SP. Wait for one clock cycle.
4. Set Read1 to SP. Verify that Out1 changes to 7 instantly.
5. Set Value to 8. Set Addr to SP+1. Wait for a clock cycle.
6. Set Read1 to SP, Read2 to SP+1. Verify that Out1, Out2 changes to 7, 8 instantly.

δ 4-to-1 Multiplexer (Mux_4Input_tb.v)
1. Set 4 inputs to 4 distinct values.
2. Set select bits to 00. Change all inputs every 50ns. Verify the output is equal to the first input at every change.
3. Set select bits to 01. Change all inputs every 50ns. Verify the output is equal to the second input at every change.
4. Set select bits to 10. Change all inputs every 50ns. Verify the output is equal to the third input at every change.
5. Set select bits to 11. Change all inputs every 50ns. Verify the output is equal to the fourth input at every change.

δ 2-to-1 Multiplexer (Mux_2Input_tb.v)
1. Set 2 inputs to 2 distinct values.
2. Set select bit to 0. Change both inputs every 50ns. Verify the output is equal to the first input at every change.
3. Set select bit to 1. Change both inputs every 50ns. Verify the output is equal to the second input at every change.

δ ALU (ALU_tb.v)
1. Set 3bit control signal to 000
2. Set two inputs to two distinct values.
3. Verify that the 16-bit output is equal to the binary 16-bit addition of the two inputs.
4. Change the two inputs every 50ns for an adequate number of cases to ensure that the ALU operation is working as intended.
5. Cycle through all previous steps, incrementing the 3-bit control up through 101 with the expectation that each output will be relative to the command specified by the control bits (000: add, 001: subtract, 010: shift left, 011: shift right, 100: or, 101: and)

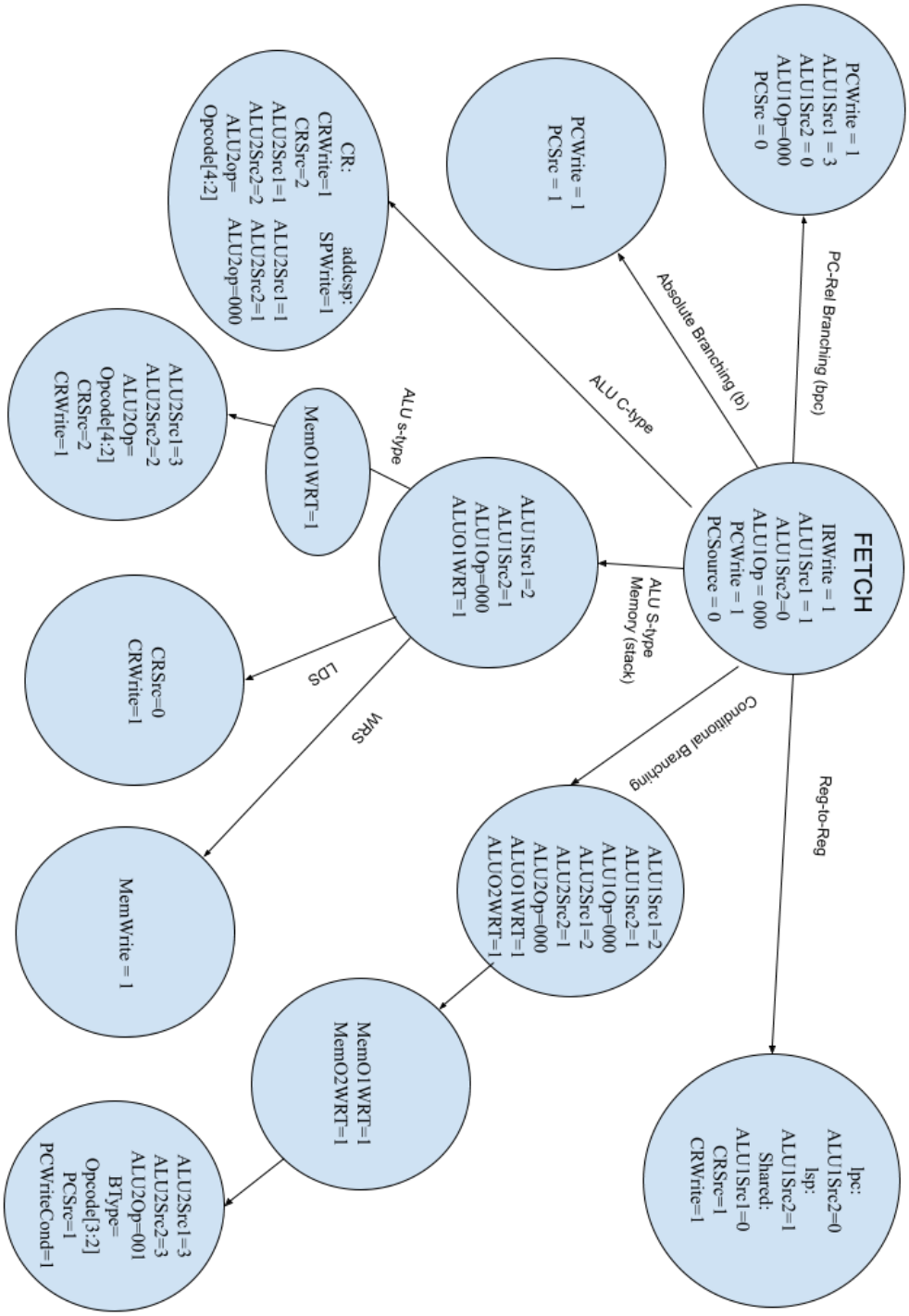
Integration Plans

- Combine PC, Instruction memory, IR, control unit, ALU1, and the corresponding two muxes to form a subsystem (Fetch.v). Test that when PC is initialized and incremented, the correct instruction is fetched and the correct control signals are generated. This is tested through Fetch_tb.v.
- Add other components to build the complete Datapath. Test the Datapath with each different type of instruction.

System Testing

- We'll start by testing each individual instruction to make sure they all function properly.
- After that, we'll test some simple code like for loops and conditional statements.
- We'll then test the complete RelPrime code.

Finite State Machine Diagram



The last state of each instruction goes back to FETCH

Euclid's Algorithm Performance Data

Test (decimal)	Test (hex)	Output (decimal)	Cycles	Time Total
2310	0x0906	13	142351	2.25ms
5040	0x13B0	11	286007	4.52ms
30030	0x754E	17	2053207	32.4ms

Unique Features

- δ We used a D-Latch instead of a register for our Instruction "Register" to save a clock cycle and combine our fetch and decode stages. Had we used a register, we would be trying to read from instruction memory and updating IR on the same clock edge, and IR wouldn't be updated properly since memory is slower. Using a latch allows IR to be updated on high CLK signal after PC gets updated on the rising edge and we finish reading from memory.
- δ We use dual-port distributed memory for our stack, which lets us read new outputs right when an input changes and read from two addresses at once.
- δ We wanted to allow for efficient conditional branching within an accumulator architecture. Our design contains dual-port memory, along with a second ALU allowing us to read both values for conditional branching in one cycle rather than spending two cycles in the memory stage.

Extra Features

- δ We designed an assembler program in Java to convert assembly code into binary and hexadecimal machine code. It is compiled into a single executable file and supports branch calculations, pseudo instructions, and debugging features.
- δ We provided a condensed version of this design document as a green sheet so that users have a condensed reference document to use while programming with our processor.

Concluding Thoughts

The process of designing and implementing a processor with its corresponding assembly language is something we never thought that we would do, and the process of designing, implementing, testing, and optimizing was intellectually stimulating. Although we doubt this is a project we will ever return too, we think it will provide a good base of knowledge if we ever embark on architecture design in a professional environment.

If we would ever return to this architecture, there a few things that it lacks that we'd like to implement. Although we were satisfied with our performance metrics, we know that modern processors use pipelining and that being able to implement this would greatly improve the speed and efficiency of our processor. Our exception handling was also lacking which is a major obstacle to this processor being used in a complex implementation. A more robust IO would also be extremely useful since reading the raw memory file would be awkward for the average consumer