Description of the registers

Saturday, October 01, 2016 7:16 PM

All register values will be 16-bits wide
$b - Temporary storage for holding on to an extra value
$flags - Assorted flags (also pages)
    15-11 = Memory Page
    …
    2 = Greater Than Flag
    1 = Less Than Flag
    0 = Equals Flag
$inport - Input register
$outport - Output register
$pc - Program Counter, not writable or readable by user
$r - The accumulator register, used implicitly
$sp - Stack Pointer
$ra - Return Address

Explanation of the procedure call conventions

Saturday, October 01, 2016 7:21 PM

When you call a subroutine, you put the first argument in the working register and the rest on the stack.

All arguments on the stack **must** be removed from the stack by the subroutine. That is to say that the entire stack must be consumed and returned to the state it was before it was called + any additional return arguments. The stack pointer should be at the top of the stack.

The first return value of a subroutine must be put in the working register and the rest of the return values get put on the stack.

When a subroutine is finished, the only changes to the stack should be the removal of arguments and the addition of return values. The caller can then move the stack pointer for any additional return values the function may have returned.

Any subroutines must always use a consistent number of arguments and return a consistent number of return values.

Before you put arguments on the stack for a procedure call, the caller must put the return address on the stack using the PUSHRA instruction.

English description of each machine language instruction format

Saturday, October 01, 2016 7:22 PM

Instructions that use numbers (I-Type)	[OP(4 bits) | Immediate(12 bits)]
Instructions that use small numbers (F-Type)	[OP(4 bits) | Function(4 bits) | Immediate(8 bits)]
Instructions that use no numbers (U-Type)	[OP(4 bits) | Function(4 bits) | Unction(8 bits)]

The I-type instructions take a 12-bit immediate values and an op-code.

The F-type instructions use the op-code:0000 and take an additional function code. They also use an 8-bit immediate value.

The U-type instructions use the op-code:0000, and take an additional function code of 0000. They also take another 8-bit "unction" code

English description of the syntax and semantics of each instruction

Saturday, October 01, 2016 7:23 PM

Instruction	Operand	Type	Description
LUI	Upper 4 bits of 16-bit immediate	F-Type	$r = immediate << 12
PAGE	Current memory page	F-Type	$flags[15:11] = immediate[numOfBits]
SKIPNIF	2 bit immediate	F-Type	Skips the next line if $flags[2:0]!=cond
SKIPIF	2 bit immediate	F-Type	Skips the next line if $flags[2:0]==cond
SLL	Amount to shift by (4 bits)	F-Type	$r = $r << immediate
SRL	Amount to shift by (4 bits)	F-Type	$r = $r >> immediate
ADDI	12-bit immediate	I-Type	$r = $r + immediate
ADDM	12-bit immediate	I-Type	$r = $r + Mem[immediate]
ANDI	12-bit immediate	I-Type	$r = $r & immediate
ANDM	12-bit immediate	I-Type	$r = $r & Mem[immediate]
CALL	12-bit immediate	I-Type	Goes to immediate+(PC+1) and puts $PC+1 into $ra
CMPI	12-bit immediate	I-Type	Compare $r and immediate and update flags
CMPM	12-bit immediate	I-Type	Compare $r and Mem[immediate] and update flags
JUMPI	12-bit immediate	I-Type	Goes to immediate+(PC+2)
LOAD	12-bit immediate	I-Type	$r = Mem[immediate]
ORI	12-bit immediate	I-Type	$r = $r | immediate
ORM	12-bit immediate	I-Type	$r = $r | Mem[immediate]
STORE	12-bit immediate	I-Type	Mem[immediate] = $r
SUBM	12-bit immediate	I-Type	$r = $r - Mem[immediate]
XORI	12-bit immediate	I-Type	$r = $r ^ immediate
XORM	12-bit immediate	I-Type	$r = $r ^ Mem[immediate]
ADDSTACK		U-Type	$r = $r + Mem[$sp]
BACKUP		U-Type	Moves $r into $b
CMPSP		U-Type	Compare $r with Mem[$sp] and update flags
DEC		U-Type	$r--
INC		U-Type	$r++
JUMP		U-Type	Goes to $r+(PC+1)
LOADSTACK		U-Type	$r = Mem[$sp]
LSP		U-Type	$r = $sp
NOOP		U-Type	NOOT NOOT (No op)
NOT		U-Type	$r = ~$r
POP		U-Type	$r = Mem[$sp] and $sp = $sp - 1
PUSH		U-Type	Mem[$sp] = $r and $sp = $sp + 1

RESTORE		U-Type	Moves \$b into \$r
RET		U-Type	Goes to \$ra
SKIPCLR		U-Type	Skips the next line when \$r == 0
SKIPSET		U-Type	Skips the next line when \$r != 0
SSP		U-Type	\$sp = \$r
STORESTACK		U-Type	Mem[\$sp] = \$r
SUBSTACK		U-Type	\$r = \$r - Mem[\$sp]
INCSP		U-Type	\$sp++
DECSP		U-Type	\$sp--
PUSHRA		U-Type	Mem[\$sp] = \$ra and \$sp = \$sp + 1
POPRA		U-Type	\$ra = Mem[\$sp] and \$sp = \$sp - 1
INPORT		U-Type	Reads data from input and puts it into \$inport
OUTPORT		U-Type	Writes data to output from \$outport

The rule for translation each assembly instruction into machine language

Saturday, October 01, 2016 7:23 PM

Instruction	Operand15	Type	Bit Pattern (little endian)
LUI	Upper 4 bits of 16-bit immediate	F-Type	0000|0001|0000[3:0 immediate]
PAGE	Page of memory we're on	F-Type	0000|0010|[7:0 immediate]
SKIPNIF	Lower 2 bits	F-Type	0000|0011|00000[2:0 immediate]
SKIPIF	Lower 2 bits	F-Type	0000|0111|00000[2:0 immediate]
SLL	Amount to shift by (4 bits)	F-Type	0000|0100|0000[3:0 immediate]
SRL	Amount to shift by (4 bits)	F-Type	0000|0110|0000[3:0 immediate]
ADDI	12-bit immediate	I-Type	0001|[11:0 immediate]
ADDM	12-bit immediate	I-Type	0010|[11:0 immediate]
ANDI	12-bit immediate	I-Type	0011|[11:0 immediate]
ANDM	12-bit immediate	I-Type	0100|[11:0 immediate]
CALL	12-bit immediate	I-Type	0101|[11:0 immediate]
CMPI	12-bit immediate	I-Type	0110|[11:0 immediate]
CMPM	12-bit immediate	I-Type	0111|[11:0 immediate]
JUMPI	12-bit immediate	I-Type	1000|[11:0 immediate]
LOAD	12-bit immediate	I-Type	1001|[11:0 immediate]
ORI	12-bit immediate	I-Type	1010|[11:0 immediate]
ORM	12-bit immediate	I-Type	1011|[11:0 immediate]
STORE	12-bit immediate	I-Type	1100|[11:0 immediate]
SUBM	12-bit immediate	I-Type	1101|[11:0 immediate]
XORI	12-bit immediate	I-Type	1110|[11:0 immediate]
XORM	12-bit immediate	I-Type	1111|[11:0 immediate]
ADDSTACK		U-Type	0000|0000|00010001
BACKUP		U-Type	0000|0000|00000001
CMPSP		U-Type	0000|0000|00001110
DEC		U-Type	0000|0000|00000010
DECSP		U-Type	0000|0000|00010100
INC		U-Type	0000|0000|00000011
INCSP		U-Type	0000|0000|00010011
JUMP		U-Type	0000|0000|00000100
LOADSTACK		U-Type	0000|0000|00001010
LSP		U-Type	0000|0000|00001011
NOOP		U-Type	0000|0000|00000000
NOT		U-Type	0000|0000|00000110
POP		U-Type	0000|0000|00010000
PUSH		U-Type	0000|0000|00001111

RESTORE		U-Type	0000|0000|00000101
RET		U-Type	0000|0000|00000111
SKIPCLR		U-Type	0000|0000|00001000
SKIPSET		U-Type	0000|0000|00001001
SSP		U-Type	0000|0000|00001101
STORESTACK		U-Type	0000|0000|00001100
SUBSTACK		U-Type	0000|0000|00010010
PUSHRA		U-Type	0000|0000|00011000
POPRA		U-Type	0000|0000|00010111
INPORT		U-Type	0000|0000|11111110
OUTPORT		U-Type	0000|0000|11111111

Example assembly language program demonstrating that your instruction into machine language

Saturday, October 01, 2016 7:25 PM

```
int relPrime(int n) {
   int m;
m = 2;
while (gcd(n, m) != 1) {
     m = m + 1;
   }
return m;
}

int gcd(int a, int b) {
  if (a == 0) {
    return b;
  }
while (b != 0) {
    if (a > b) {
      a = a - b;
    } else {
      b = b - a;
    }
  }
return a;
}
```

```
INPORT      // grab the input
PUSH        // put the input on the stack for future use, not as a parameter yet
BACKUP      // backup the input
ANDI 0      // clear out working register
ADDI 2      // set working register to 2
PUSH        // put 2 on the stack
PUSH        // put it on the stack again, this time as a parameter
RESTORE     // put input back in working register

gcdLoop:    // keeps computing GCD and incrementing m if gcd isn't 1
CALL gcd    // compute GCD
CMPI 1      // compare working register against 1
SKIPNIF 1   // skip if working not 1 (flags aren't 001)
JUMPI end   // we found a relatively prime number. go to the end
POP
INC
PUSH
PUSH
DECSP
DECSP
LOADSTACK
INCSP
INCSP
JUMPI gcdLoop   // compute gcd again


gcd:
SKIPCLR     // if a!=0 go to the while
JUMPI while // (2)
POP         // pop b off the stack
RET         // return b

while:      // a should be in working register
BACKUP      // backup a
LOADSTACK   // put b in $r
CMPI 0      // compare with 0
SKIPIF 1    // Skip the jump if b==0
JUMPI continue  // Jump to a>b (3)
POP         // b==0 so pop b off the stack
RESTORE     // put a back in $r
RET         // return a

continue:   // a should be in $at
RESTORE     // get a back in $r
CMPSP       // compare a against b
SKIPIF 4    // if a>b skip the jump
JUMPI lte   // jump to else case (2)
SUBSTACK    // subtract b from a
JUMPI while // go back to while condition (-14)

lte:        // less than or equal to case, a should be in $r
PUSH        // put a on the stack
DECSP       // decrement stack pointer
LOADSTACK   // get b in $r
INCSP       // increment stack pointer
SUBSTACK    // subtract a from b
DECSP       // decrement stack pointer
STORESTACK  // update b in the stack
INCSP       // increment stack pointer
POP
JUMPI while // (-24)
NOOP


end:
POP         // grab the answer from the stack
OUTPORT     // output the answer
```

Assembly language fragments for common operations

Saturday, October 01, 2016 7:26 PM

Simple comparisons

If (a < b)
   GOTO label

```
LOAD A
CMPM B
SKIPNIF 2
JUMPI label
```

If (a <= b)
   GOTO label

```
LOAD A
CMPM B
SKIPNIF 3
JUMPI label
```

If (a == b)
   GOTO label

```
LOAD A
CMPM B
SKIPNIF 1
JUMPI label
```

If (a > b)
   GOTO label

```
LOAD A
CMPM B
SKIPNIF 4
JUMPI label
```

If (a >= b)
   GOTO label

```
LOAD A
CMPM B
SKIPIF 2
JUMPI label
```

If (a != b)
   GOTO label

```
LOAD A
CMPM B
SKIPIF 1
JUMPI label
```

Procedure Call

```
a = adder(a,b)
int adder(int a, int b) {
   return a+b;
}
```

```
PUSHRA
LOAD B
PUSH
LOAD A
CALL adder
POPRA
STORE A

adder:
    ADDSTACK
    BACKUP
    POP
    RESTORE
    RET
```

While Loop

```
while(a>b) {
   noop;
   a--;
}
```

```
LOAD A
loop:
    CMPM B
    SKIPIF 0
    JUMPI finish
    NOOP
    DEC
    JUMPI loop
```

Machine language translations of the programs written for testing

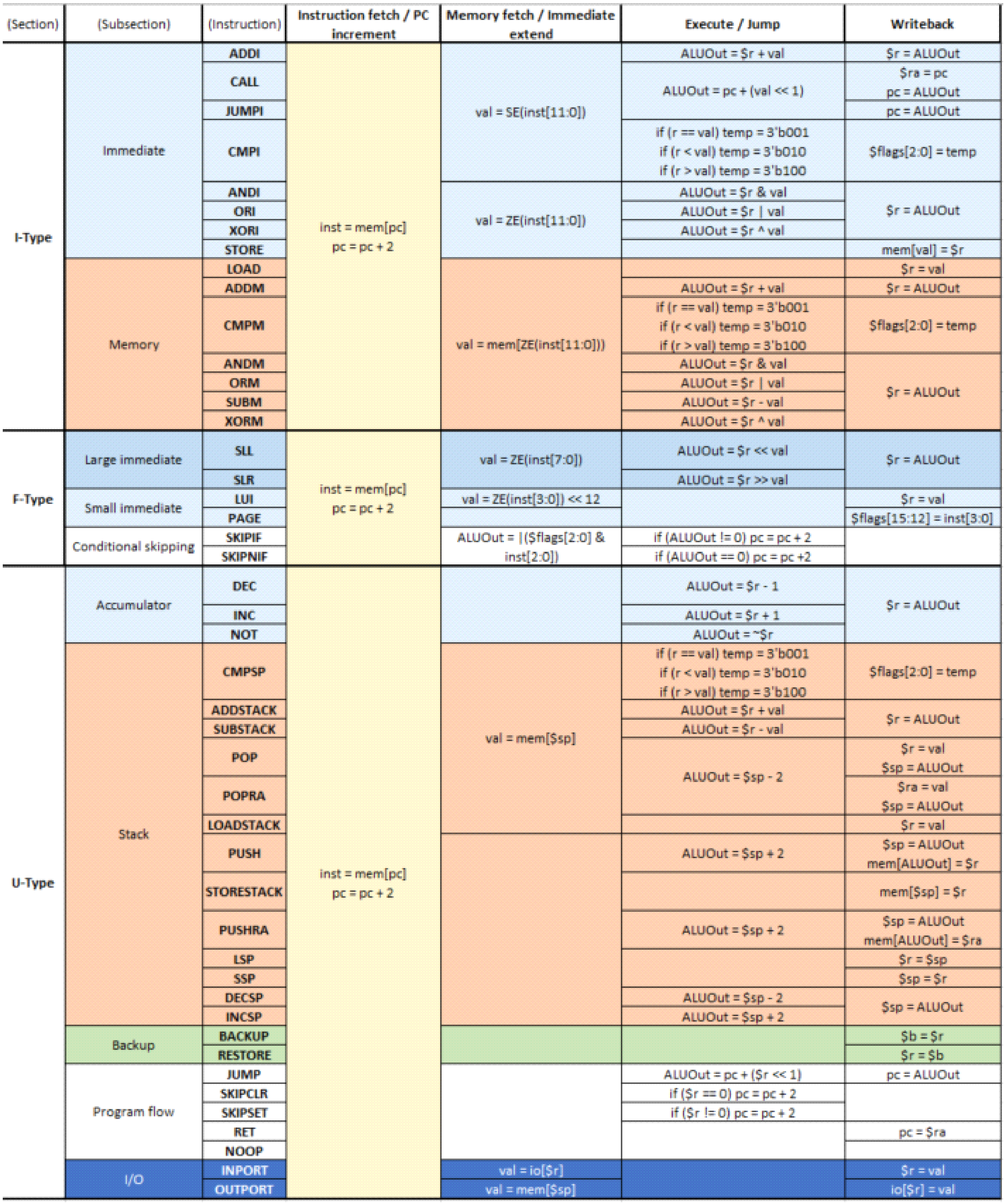
Saturday, October 01, 2016 7:26 PM

Relprime code:

```
0000 0100 a06e 000d 8000 00fe 000f 0001
3000 1002 000f 000f 0005 500d 6001 0301
8027 0010 0003 000f 000f 0014 0014 0013
0013 8ff2 0008 8002 0010 0007 0001 6000
0701 8003 0010 0005 0007 0005 000e 0704
8002 0012 8ff2 000f 0014 0013 0012 0014
0013 0010 8fe8 0000 0010 00ff 8fff
```

Unified RTL

Monday, October 24, 2016 9:04 AM



(Section)	(Subsection)	(Instruction)	Instruction fetch / PC increment	Memory fetch / Immediate extend	Execute / Jump	Writeback
I-Type	Immediate	ADDI	inst = mem[pc] pc = pc + 2	val = SE(inst[11:0])	ALUOut = $r + val	$r = ALUOut
		CALL			ALUOut = pc + (val << 1)	$ra = pc pc = ALUOut
		JUMPI				pc = ALUOut

		CMPI			if (r == val) temp = 3'b001
					if (r < val) temp = 3'b010
					if (r > val) temp = 3'b100
						$flags[2:0] = temp

		ANDI		val = ZE(inst[11:0])	ALUOut = $r & val	$r = ALUOut
		ORI			ALUOut = $r | val	
		XORI			ALUOut = $r ^ val	
		STORE				mem[val] = $r
	Memory	LOAD		val = mem[ZE(inst[11:0])]		$r = val
		ADDM			ALUOut = $r + val	$r = ALUOut

		CMPM			if (r == val) temp = 3'b001
					if (r < val) temp = 3'b010
					if (r > val) temp = 3'b100
						$flags[2:0] = temp

		ANDM			ALUOut = $r & val	$r = ALUOut
		ORM			ALUOut = $r | val	
		SUBM			ALUOut = $r - val	
		XORM			ALUOut = $r ^ val	
F-Type	Large immediate	SLL	inst = mem[pc] pc = pc + 2	val = ZE(inst[7:0])	ALUOut = $r << val	$r = ALUOut
		SLR			ALUOut = $r >> val	
	Small immediate	LUI		val = ZE(inst[3:0]) << 12		$r = val
		PAGE				$flags[15:12] = inst[3:0]
	Conditional skipping	SKIPIF		ALUOut = |($flags[2:0] & inst[2:0])	if (ALUOut != 0) pc = pc + 2	
		SKIPNIF			if (ALUOut == 0) pc = pc +2	
U-Type	Accumulator	DEC	inst = mem[pc] pc = pc + 2		ALUOut = $r - 1	$r = ALUOut
		INC			ALUOut = $r + 1	
		NOT			ALUOut = ~$r	

	Stack	CMPSP		val = mem[$sp]	if (r == val) temp = 3'b001
					if (r < val) temp = 3'b010
					if (r > val) temp = 3'b100
						$flags[2:0] = temp

		ADDSTACK			ALUOut = $r + val	$r = ALUOut
		SUBSTACK			ALUOut = $r - val	
		POP			ALUOut = $sp - 2	$r = val $sp = ALUOut
		POPRA				$ra = val $sp = ALUOut
		LOADSTACK				$r = val
		PUSH			ALUOut = $sp + 2	$sp = ALUOut mem[ALUOut] = $r
		STORESTACK				mem[$sp] = $r
		PUSHRA			ALUOut = $sp + 2	$sp = ALUOut mem[ALUOut] = $ra
		LSP				$r = $sp
		SSP				$sp = $r
		DECSP			ALUOut = $sp - 2	$sp = ALUOut
		INCSP			ALUOut = $sp + 2	
	Backup	BACKUP				$b = $r
		RESTORE				$r = $b
	Program flow	JUMP			ALUOut = pc + ($r << 1)	pc = ALUOut
		SKIPCLR			if ($r == 0) pc = pc + 2	
		SKIPSET			if ($r != 0) pc = pc + 2	
		RET				pc = $ra
		NOOP				
	I/O	INPORT		val = io[$r]		$r = val
		OUTPORT		val = mem[$sp]		io[$r] = val

List of Components

Tuesday, October 11, 2016 6:56 PM

- Arithmetic Logic Unit
  - Inputs
    - Operand A (16-bit)
    - Operand B (16-bit)
    - Control (4-bit)
      - 0 - Add A+B
      - 1 - Sub A-B
      - 2 - Add A+B, shift result left by 12
      - 3 - Set should_skip to 1 if skipnif condition is met
      - 4 - Set should_skip to 1 if skipif condition is met
      - 5 - Shift A left by B amount
      - 6 - Shift A right by B amount
      - 7 - A & B
      - 8 - A | B
      - 9 - A ^ B
      - 10 - ~A
      - 11 - Compare A and B and set the flags accordingly
      - 12 - Set should_skip to 1 if A is 0
      - 13 - Set should_skip to 1 if A is not 0
  - Outputs
    - Result (16-bit)
    - `should_skip` Control Bit (1-bit)
    - Comparison Flags (3-bit)
  - Does math. Takes operand A and B and performs an operation determined by the control signal (straight from the control unit.) These operations include and, or, xor, not, add, subtract, and comparison. Puts the output on result, indicates if we should skip using the should_skip indicator, and sets the comparison flags if needed.
  - How to implement: In verilog, use a switch for the control bits that determines which operation should be done.
  - Tests:
    - AND, OR, XOR
      - Input domain:
        - Control ∈ {0x7, 0x8, 0x9}
        - A,B ∈ {0x0000, 0x0001, …, 0xFFFF}
      - Output ranges we care about in this case:
        - R ∈ {0x0000, 0x0001, …, 0xFFFF}
      - Boundary value analysis:
        - For all combinations of A and B where A, B ∈ {0x0000, 0xFFFF},
          if Control = 0x7, assert that R = A & B,
          if Control = 0x8, assert that R = A | B,
          if Control = 0x9, assert that R = A ^ B.
    - NOT:
      - Input domain:
        - Control = 0xA
        - A, B ∈ {0x0000, 0x0001, …, 0xFFFF}
      - Output ranges we care about in this case:
        - R ∈ {0x0000, 0x0001, …, 0xFFFF}
      - Boundary value analysis:
        - For all combinations of A and B where A, B ∈ {0x0000, 0xFFFF},

        - assert that R = ~A
- Add:
    - Input domain:
        - Control = 0x0
        - A, B ∈ {-32768, -32767, -32766 …, 32766, 32767}
    - Output ranges we care about in this case:
        - R ∈ {-32768, -32767, -32766 …, 32766, 32767}
    - Boundary value analysis:
        - For all combinations of A and B where A, B ∈ {-32768, 32767} and A ≠ B, assert that R = A + B

        - For all combinations of A and B where A, B ∈ {32767, 1} and A ≠ B, assert that R = -32768

        - For all combinations of A and B where A, B ∈ {-32768, -1} and A ≠ B, assert that R = 32767
- Subtract:
    - Input domain:
        - Control = 0x1
        - A, B ∈ {-32768, -32767, -32766 …, 32766, 32767}
    - Output ranges we care about in this case:
        - R ∈ {-32768, -32767, -32766 …, 32766, 32767}
    - Boundary value analysis:
        - For all combinations of A and B where A, B ∈ {-32768, 32767} and A = B, assert that R = 0

        - For all combinations of A and B where A, B ∈ {32767, -1} and A ≠ B, assert that R = -32768

        - Let A = -32768 and B = 1
          assert that R = 32767

        - Let A = 1 and B = -32768
          assert that R = -32767
- Comparison:
    - Input domain:
        - Control = 0xB
        - A, B ∈ {-32768, -32767, -32766 …, 32766, 32767}
    - Output ranges we care about in this case:
        - Comparison Flags ∈ {0x1, 0x2, 0x4}
    - Boundary value analysis:
        - For all combinations of A and B where A, B ∈ {-32768, 32767} and A = B assert that Comparison Flags = 0x1

        - Let A = -32768 and B = 32767
          assert that Comparison Flags = 0x2

        - Let A = 32767 and B = -32768
          assert that Comparison Flags = 0x4
- Skipif and Skipnif
    - Input domain:
        - Control ∈ {0x3, 0x4}
        - A,B ∈ {0x0000, 0x0001, …, 0xFFFF}

            - Output ranges we care about in this case:
                - Should skip ∈ {0, 1}
            - Testing:
                - For all combinations of A and B,
                  Let i be a bit on a bus and let i ∈ {0, 1, 2}
                  assert that Should skip = 1 if Control = 0x4 and if for any i, A_i = 1 and B_i = 1. Should skip = 0 otherwise
                  assert that Should skip = 0 if Control = 0x3 and if for any i, A_i = 1 and B_i = 1. Should skip = 1 otherwise

- Control Unit
    - Input
        - Instruction (16-bit)
    - Outputs (see next page for details)
        - r_src (1-bit)
        - ra_src (1-bit)
        - ALU_src_A (3-bit)
        - ALU_src_B (4-bit)
        - mem_shift (1-bit)
        - memory_write (1-bit)
        - memory_save_src (2-bits)
        - memory_addr_src (1-bit)
        - memory_sp_dec (1-bit)
        - memory_mod_sp (1-bit)
        - should_jump (1-bit)
        - r_write (1-bit)
        - r_backup (1-bit)
        - r_restore (1-bit)
        - page_write (1-bit)
        - cmp_write (1-bit)
        - sp_write (1-bit)
        - ra_write (1-bit)
        - ALUOp (4-bit)
    - The control unit controls the ALU and the various multiplexors that direct the path of data through the processor. The decision of how to set each output comes from the opcode. This is a very complex part and may God have mercy on our souls.
    - To test, give the control unit each valid instruction and compare the outputted control bits to what they should be.

- Register File
    - Inputs
        - Control:
            - R Write (1-bit)
            - Page Write (1-bit)
            - Compare Write (1-bit)
            - Stack Pointer Write (1-bit)
            - Return Address Write (1-bit)
            - Clock (1-bit)
        - Data:
            - R Input (16-bit)
            - Page Input (4-bit)
            - Compare Input (3-bit)
            - Stack Pointer Input (16-bit)
            - Return Address Input (16-bit)

        - Functionality:
            - R Backup (1-bit)
            - R Restore (1-bit)
    - Outputs
        - R Output (16-bit)
        - Flag Output (16-bit)
        - Stack Pointer Output (16-bit)
        - Return Address Output (16-bit)

    - The register file remembers values. The output of the registers is always on the outputs, but to write a value, you have to put a value on the value pin, and the high value on the corresponding write pin.
    - To implement, create inputs for all the write control bits and write data. Have outputs as described above. Internally, create registers for r, backup, page, compare, sp, and ra and pass the clock, and the corresponding write control and data to each one. Pass the outputs from the registers to the corresponding output busses. On the clock edge, the internal registers should replace their stored value with what is contained in their data input bus ONLY IF the control bit for that register is set. The registers should consistently output what is stored within them to their corresponding output busses.
    - To test, save and read from every register. Try saving a value with the write enabled and disabled to see if it only saves when the write pin is enabled. Then check to see that the value was actually saved on the next cycle.

- PC Register
    - Inputs
        - New PC (16-bit)
        - Clock (1-bit)
    - Outputs
        - Current PC (16-bit)
    - PC always outputs the current PC. The PC will be updated when the clock is high.
    - To implement, create inputs/outputs as described above, and update the PC on the clock edge.
    - To test, put a value in it and make sure it persists over the clock cycle.

- PC Incrementer
    - Inputs
        - Old PC (16-bit)
    - Output
        - New PC (16-bit)
    - The whole point of this is to increment the old PC by two.
    - To implement, create inputs/outputs as described above, and always output the input +2
    - To test, feed the incrementer several values and make sure that that value plus two comes out the other side.
- Memory
    - Inputs
        - Address (16-bit)
        - Memory Input (16-bit)
        - Memory Write (1-bit)
    - Outputs
        - Memory Output (16-bit)
    - Memory's output is always the value at the address provided on the address pin. To write to it, you specify the address, the value you want to store, and set the write pin to high.
    - To implement, configure inputs/outputs as described above and write/read to memory as

the control bits indicate. On the FPGA, we can only support 10-bits of memory, so in practice this bus will only be 10-bits. This dual-port memory will be Xilinx synthesized block memory.
    - To test, make sure that each memory address will hold a value for as long as the processor is on. Check and make sure that memory writes only when the write bit is enabled.

- Multiplexors
    - See next page for multiplexor definitions
- Logic Gates
- Nets
- Sign Extender
    - Input
        - Value (some number of bits)
    - Output
        - Value (16-bit)
    - The output will be the sign extended value, extended to 16-bits.
    - To implement, read the value of the MSB of the input and set any higher bits to that value.
    - To test, give it a bunch of negatives and a bunch of positives and make sure it/ sign extends the numbers properly.

- Zero Extender
    - Input
        - Value (some number of bits)
    - Output
        - Value (16-bit)
    - The output will be the zero extended value, extended to 16-bits.
    - To implement, configure inputs/outputs as described above and set higher bits to 0.
    - To test, make sure the bits added are all zero. Feed it multiple sizes of data to be sure.

Integration Plan

Wednesday, October 19, 2016 2:15 PM

Memory Subsystem:
Description: Connect the 3 memory muxes (memory_sp_dc, memory_addr_src, memory_save_src), the memory unit, and the stack-pointer adder with appropriate wires/buses.
Tests: Iterate through all combination of control signals for the muxes and for each combination try saving various values to various addresses in memory. Also, for each combination check to see if what is being read from memory is what is expected.

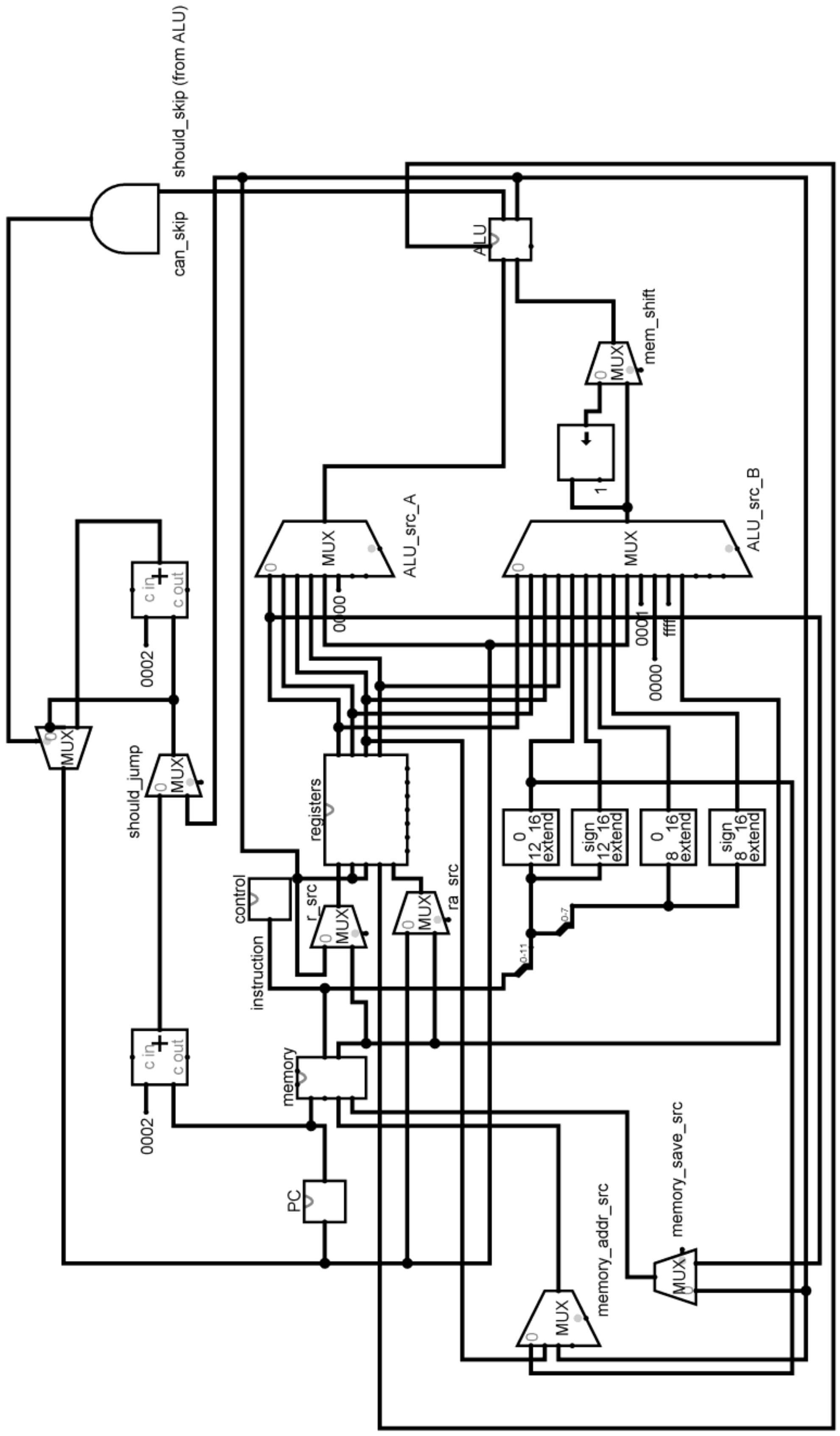
PC Subsystem:
Description: Connect the 2 PC muxes and the 2 PC adders with appropriate wires/buses.
Tests: Iterate through all combinations of control signals, and for each combination check to see if PC is being updated in the expected manner (stepping, skipping, or jumping).

ALU Subsystem:
Description: Connect the program memory, register file, sign extenders, zero extenders, and the ALU source muxes
Tests: Iterate through all combinations of control signals, manually supplying values to represent memory outputs, and make sure the ALU outputs what is expected. Also make sure that the correct registers get written to and that they receive the correct data.

Overall:
Description: Connect the 3 previously described subsystems
Tests: Give the datapath several sets of control signals that we know may result from a real instruction, and make sure at the end of the cycle that memory and any registers have changed as we expect them to.

CPU Datapath

Wednesday, October 19, 2016 2:43 PM



Datapath Multiplexor Value Descriptions

Tuesday, October 18, 2016 12:30 AM

- r_src
  - 0 - Set R from ALU
  - 1 - Set R from memory out
- ra_src
  - 0 - Set RA from PC
  - 1 - Set RA from memory out
- ALU_src_A
  - 0 - R
  - 1 - FLAG
  - 2 - SP
  - 3 - RA
  - 4 - New PC
  - 5 - 0x0000
- ALU_src_B
  - 0 - R
  - 1 - FLAG
  - 2 - SP
  - 3 - RA
  - 4 - Zero Extended 11:0 immediate
  - 5 - Sign Extended 11:0 immediate
  - 6 - Zero Extended 7:0 immediate
  - 7 - Sign Extended 7:0 immediate
  - 8 - New PC
  - 9 - 0x0001
  - 10 - 0x0000
  - 11 - 0xFFFF
  - 12 - Memory Out
- memory_write
  - 0 - Do not write to memory
  - 1 - Write to memory
- memory_save_src
  - 0 - Data to save comes from ALU
  - 1 - Data to save comes from R
  - 2 - Data to save comes from RA
- memory_addr_src
  - 0 - Address comes from zero extended 11:0 immediate
  - 1 - Address comes from SP
  - 2 - Address comes from ALU
- should_jump
  - 0 - Do not set PC from ALU
  - 1 - Set PC from ALU
- can_skip
  - 0 - Do not skip, even if ALU says to
  - 1 - Skip if the ALU wants to skip
- r_write
  - 0 - Do not change R
  - 1 - Write r_in to R
- r_backup
- 0 - Do not change B
  - 1 - Move R to B in the register file

- r_restore
    - 0 - Do not change R
    - 1 - Move B to R in the register file
- page_write
    - 0 - Do not change the page portion of the flag
    - 1 - Change the page portion of the file according to page_in
- cmp_write
    - 0 - Do not change the comparison flag portion of the flag
    - 1 - Change the comparison flag portion of the file according to cmp_in
- sp_write
    - 0 - Do not change SP
    - 1 - Write sp_in to SP
- ra_write
    - 0 - Do not change RA
    - 1 - Write ra_in to RA
- ALUOp
    - 0 - Add A+B
    - 1 - Sub A-B
    - 2 - Add A+B, shift result left by 12
    - 3 - Set should_skip to 1 if skipnif condition is met
    - 4 - Set should_skip to 1 if skipif condition is met
    - 5 - Shift A left by B amount
    - 6 - Shift A right by B amount
    - 7 - A & B
    - 8 - A | B
    - 9 - A ^ B
    - 10 - ~A
    - 11 - Compare A and B and set the flags accordingly
    - 12 - Set should_skip to 1 if A is 0
    - 13 - Set should_skip to 1 if A is not 0

Control Signals Per Instruction

Tuesday, October 25, 2016 4:21 PM

Instruction	r_src	ra_src	ALU_src_A	ALU_src_B	memory_write	memory_save_sr	memory_addr_src	can_skip	should_jump	r_write	r_backup	r_restore	page_write	cmp_write	sp_write	ra_write	ALUOp
LUI	0	x	5	6	0	x	x	0	0	1	0	0	0	0	0	0	2
PAGE	x	x	5	6	0	x	x	0	0	0	0	0	1	0	0	0	0
SKIPNIF	x	x	1	6	0	x	x	1	0	0	0	0	0	0	0	0	3
SKIPIF	x	x	1	6	0	x	x	1	0	0	0	0	0	0	0	0	4
SLL	0	x	0	6	0	x	x	0	0	1	0	0	0	0	0	0	5
SRL	0	x	0	6	0	x	x	0	0	1	0	0	0	0	0	0	6
ADDI	0	x	0	5	0	x	x	0	0	1	0	0	0	0	0	0	0
ADDM	0	x	0	12	0	x	0	0	0	1	0	0	0	0	0	0	0
ANDI	0	x	0	4	0	x	x	0	0	1	0	0	0	0	0	0	7
ANDM	0	x	0	12	0	x	0	0	0	1	0	0	0	0	0	0	7
CALL	x	0	4	5	0	x	x	0	1	0	0	0	0	0	0	1	0
CMPI	0	x	0	5	0	x	x	0	0	0	0	0	0	1	0	0	11
CMPM	0	x	0	12	0	x	0	0	0	0	0	0	0	1	0	0	11
JUMPI	x	0	4	5	0	x	x	0	1	0	0	0	0	0	0	0	0
LOAD	1	x	x	x	0	x	0	0	0	1	0	0	0	0	0	0	x
ORI	0	x	0	4	0	x	x	0	0	1	0	0	0	0	0	0	8
ORM	0	x	0	12	0	x	0	0	0	1	0	0	0	0	0	0	8
STORE	x	x	0	10	1	1	0	0	0	0	0	0	0	0	0	0	x
SUBM	0	x	0	12	0	x	0	0	0	1	0	0	0	0	0	0	1
XORI	0	x	0	4	0	x	x	0	0	1	0	0	0	0	0	0	9
XORM	0	x	0	12	0	x	0	0	0	1	0	0	0	0	0	0	9
ADDSTACK	0	x	0	12	0	x	1	0	0	1	0	0	0	0	0	0	0
BACKUP	x	x	x	x	0	x	x	0	x	0	1	0	0	0	0	0	x
CMPSP	0	x	0	12	0	x	1	0	0	0	0	0	0	1	0	0	11
DEC	0	x	0	11	0	x	x	0	0	1	0	0	0	0	0	0	0
INC	0	x	0	9	0	x	x	0	0	1	0	0	0	0	0	0	0
JUMP	x	x	4	0	0	x	x	0	1	0	0	0	0	0	0	0	0
LOADSTACK	1	x	2	10	0	x	1	0	0	1	0	0	0	0	0	0	0
LSP	0	x	2	10	0	x	x	0	0	1	0	0	0	0	0	0	0
NOOP	x	x	5	10	0	x	x	0	0	0	0	0	0	0	0	0	0
NOT	0	0	0	x	0	x	x	0	0	1	0	0	0	0	0	0	10
POP	1	x	2	11	1	x	2	0	0	1	0	0	0	0	1	0	0
PUSH	x	x	2	9	1	1	2	0	0	0	0	0	0	0	1	0	0
RESTORE	x	x	x	x	0	x	x	0	x	0	0	1	0	0	0	0	x
RET	x	x	3	10	0	x	x	0	1	0	0	0	0	0	0	0	0
SKIPCLR	x	x	0	x	0	x	x	1	0	0	0	0	0	0	0	0	12
SKIPSET	x	x	0	x	1	x	x	1	0	0	0	0	0	0	0	0	13
SSP	x	x	0	10	0	x	x	0	0	0	0	0	0	0	1	0	0
STORESTACK	x	x	x	x	1	1	1	0	0	0	0	0	0	0	0	0	x
SUBSTACK	0	x	0	12	0	x	1	0	0	1	0	0	0	0	0	0	1
INCSP	x	x	2	9	0	x	x	0	0	0	0	0	0	0	1	0	0
DECSP	x	x	2	11	0	x	x	0	0	0	0	0	0	0	1	0	0
PUSHRA	x	x	3	9	1	0	2	0	0	0	0	0	0	0	1	0	0
POPRA	x	x	3	11	0	x	2	0	0	0	0	0	0	0	1	1	0

Executable Format

Wednesday, November 02, 2016 11:20 AM

A BAMF executable always includes its relevant start up code. The first thing it does is set the stack pointer and jump to the program's entry point. The program's entry point is after the global environment. These variables do not exist on the stack.

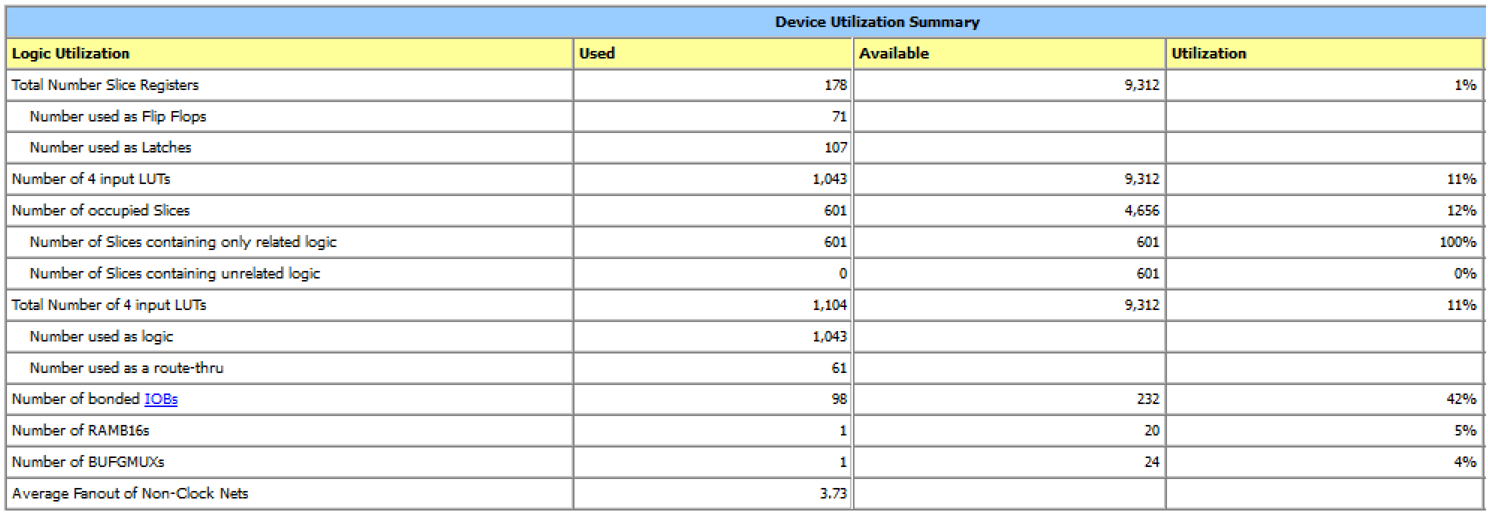
The form of the executable is as follows:

```
NOOP
LUI 0x0000
ORI <pointer to stack>
SSP
JUMPI <global environment size>
…
Global Environment
…
User code
…
JUMPI -1 #HALT Pseudo Instruction expands to this
Remaining space used for stack. SP will initially point here
```

Performance

Wednesday, November 09, 2016 2:53 PM

To store the whole program, the assembler required 110 bytes.
The total number of instructions that were executed was 102230.
The total number of cycles required to execute relPrime was 102230.
The average number of cycles is one because this is a single cycle datapath.
The cycle time for our design was 44.107ns (22.672MHz).
The total execution time was 2,044,700ns. (2.0447ms)
The device utilization summary:



Device Utilization Summary			
Logic Utilization	**Used**	**Available**	**Utilization**
Total Number Slice Registers	178	9,312	1%
Number used as Flip Flops	71		
Number used as Latches	107		
Number of 4 input LUTs	1,043	9,312	11%
Number of occupied Slices	601	4,656	12%
Number of Slices containing only related logic	601	601	100%
Number of Slices containing unrelated logic	0	601	0%
Total Number of 4 input LUTs	1,104	9,312	11%
Number used as logic	1,043		
Number used as a route-thru	61		
Number of bonded IOBs	98	232	42%
Number of RAMB16s	1	20	5%
Number of BUFGMUXs	1	24	4%
Average Fanout of Non-Clock Nets	3.73		