Processor Project

CSSE232

Cullen LaKemper
Joseph Peters
Russel Staples
Will Yelton

Table of Contents

Architecture Description

Name: CHINPO (**C**hinpo **H**ardware **I**s **N**ot **P**erfectly **O**ptimized)

About:
CHINPO is a 16-bit word, instruction, and register load store based architecture, which utilizes two preset operational registers connected to the ALU which are used for all ALU operations and temporary registers to store results. The architecture focuses on completing instructions quickly and preparing for the next operation concurrently. ALU operations such as addition and subtraction are always operated on the A and B registers and the result is placed into the destination register set through the instruction. Every non-Immediate command can concurrently move a value from a register into register A, register B, or both. Immediate commands accept an 8-bit immediate.

Registers

Number	Symbol	Description
0	$0	Zero register: Always equal to 0 cannot be changed
1	$sp	Stack Pointer: Points to the current top of the stack
2	$ra	Return Address: Points to address the current function must jump to when concluded
3	$sr	System reserved: Used for interrupts and cause etc...
4	$at	Assembler Temporary for pseudo instructions
5	$a0	Argument 0: Place and receive function arguments here
6	$a1	Argument 1: Place and receive function arguments here
7	$v0	Function Return: Place function returns here
8	$t0	Temporary register 0
9	$t1	Temporary register 1
10	$t2	Temporary register 2
11	$t3	Temporary register 3
12	$t4	Temporary register 4
13	$t5	Temporary register 5
14	$A	A: Operations register 0
15	$B	B: Operations register 1

Instruction Formats

DR Type Instructions:

4 bits	4 bits	4 bits	1 b	1 b	1 b	1 b
op	rd	rm	ma	mb	CLRa	CLRb

op	:	Operation Code	number (Defined in the table below)
rd	:	register destination	number (Addressed directly defined above)
rm	:	register to move	number (Addressed directly defined above)
ma	:	move to a	boolean (1-move, 0- do not move)
mb	:	move to b	boolean (1-move, 0- do not move)
CLRa	:	clear a	boolean (1- clear, 0- do not clear)
CLRb	:	clear b	boolean (1- clear, 0- do not clear)

I Type Instructions:

4 bits	4 bits	8 bits
op	rd	immediate

op	:	Operation code	number (Defined in the table below)
rd	:	register destination	number (Addressed directly defined above)

Instructions

DR (Double Register) Type Instructions

The register designated by rm is moved to either the A or B register as designated by ma/mb concurrently with what is described in the instruction description. Also the CLRa/CLRb bits can clear the values in A or B after an instruction is completed and before the move happens.

Syntax: inst rd, rm, ma, mb, CLRa, CLRb
Example: add $t1, $t2, 0, 1, 0, 0,

Op Code		Symbol	Name	Description
2	0010	add	Add	Adds A to B and stores in rd
6	0110	sub	Subtract	Subtracts B from A and stores in rd
0	0000	and	And	Bitwise and of A and B
1	0001	or	Or	Bitwise or of A and B
5	0101	mv	Move	Ignores the rd register
7	0111	slt	Set Less Than	If A < B set rd to 1 else set rd to 0
3	0011	jr	Jump Register	Jumps to address held in A (rd not used)

I (Immediate) Type Instructions

Values are stored in the register designated by rd. The immediate does a variety of things depending on the specific instruction.

Syntax: inst rd, im
Example: beq $t0, BRANCH
lw $t1, 4

4	0100	lli	Load Lower Immediate	Loads <immediate> into least significant bits of rd (sign extended)
9	1001	ori	Or Immediate	Bitwise or with A and zero extended <immediate>
13	1101	sll	Shift Left Logical	Shifts value in A by signed (immediate) and stores in rd, positive numbers shift left, negative numbers shift right
10	1010	addi	Add Immediate	Adds <immediate> to A and stores in rd
8	1000	j	Jump	Jumps to tag or address PC[15-9] + <immediate> + 0

11	1011	jal	Jump and Link	Jumps to tag or address PC[15-9] + <immediate> + 0 and stores the return address (PC+4) into $ra
Op Code		**Symbol**	**Name**	**Description**
14	1110	lw	Load Word	The value at the address in A + (<immediate> * 2) is stored in rd
15	1111	sw	Store Word	The value in B is stored in the address in A + (<immediate> * 2)
12	1100	beq	Branch On Equal	If A == B move <immediate> instructions Beq jumps to the address defined by the (first 7 bits of the program counter + 2) + (the 8 bit immediate given shifted once)

Input / Output

Memory Mapping

We will have a special block of memory that is dedicated to holding both the input words and the output words. There will be specific places that input and output are stored within this block. The input section can be controlled by switches or any other input device that can change bits in the memory to the desired value. When the program is done running, it will store the output in a specific memory location, then this can be displayed on lights or any other device that can show bit values.

Interrupts

In order to get the processor to know that the input value is ready, there will be one more space in the dedicated memory block, and when a bit is set to a one, it will stay one until the processor reads the input value. When that memory location is set to a one, it will trigger an interrupt which will set cause the processor to go into the exception handler, and from there it will get the input value and run Euclid's Algorithm on it and then put the output in the specified memory location.

Memory

Our instruction set will support up to 16 bit addresses for memory, but our processor will only support 9 bit addresses so we will have to have some exceptions in the processor. We will have the following blocks in memory:

Decimal Range	**Binary Range**	**Memory Type**
0-255	0000000000-0011111110	Instruction Memory
256-511	0100000000-0011111110	I/O Memory
512-1023	1000000000-1111111110	Stack Memory
1024-65535	10000000000-1111111111111110	Unused

The instruction memory will be where the program is stored in memory, users cannot access this section of memory. The next section of memory is reserved for I/O, see the section on I/O for details. Users can also not access the I/O memory data. The last section is the only place where users can access the memory. It will be used for stack storage while the program is running.

If any address is accessed that is not in the stack area, the memory unit will not give the data in that address or write to that address, it will just output zero and send an error.

Register Transfer Language for Instructions

DR-Type	**I-Type**	**lw**	**sw**	**beq**
IR = Mem[PC] PC = PC + 2				
ALUout = PC+ S/E(IR[7-0]<<1)				

ALUout = A op B
if(IR[1]==1) {
REG[1110] = 0 }
else if(IR[3]==1) {
REG[1110] = REG[IR[7-4]]
}
 	
if(IR[0]==1) {
REG[1111] = 0 }
else if(IR[2]==1) {
REG[1111] = REG[IR[7-4]]
}
	ALUout = A op S/E(IR[7-0])	ALUout = A + SE(IR[7-0]<<1)	ALUout = A + S/E(IR[7-0]<<1)	IF (REG[1110] == REG[1111])
				PC = ALUout

REG[IR[11-8] = ALUout	REG[IR[11-8]] = ALUout	MDR = MEM[ALUout]	MEM[ALUout] = REG[1111]	
		REG[IR[11-8]] = MDR		

mv	**j**	**jr**	**jal**
IR = Mem[PC] PC = PC + 2			
ALUout = PC+ S/E(IR[7-0]<<1)			

if(IR[1]==1) {
  REG[1110] = 0
}
if(IR[0]==1) {
  REG[1111] = 0
}
if(IR[3]==1) {
  REG[1110] = REG[IR[7-4]]
}
if(IR[2]==1)
  REG[1111] = REG[IR[7-4]]
}
	PC = ALUout	ALUout = REG[1110]	PC=ALUout
			ALUout=PC

		PC=ALUout	REG[0010]=ALUout

Component List

Name	Description	Inputs	Outputs	Testing
PC	Register that stores the program counter	**Data In (16 bits)**: The data input of the register **Clock (1 bit)**: The clock of the processor **Write (1 bit)**: Controls if data is being written or not **Reset (1 bit)**: Resets data in register to 0 if the clock is enabled	**Data Out (16 bits)**: The data output of the register	To test these, we need to just make sure that they can be written when the write signal is high and not written when it is low. Also, we need to make sure that it stores its value and outputs it to the data out port.
IR	Instruction register			
A	Registers that input into ALU. A is always the A register and B is either the B register or an immediate			
B				
MDR	Register that stores memory data			
ALUout	Register that stores output of ALU			
REG	Register file, it has the ability to move data to the A and B registers, as well as clear them. It always outputs the data in the A and B registers. **To Build:** We will have 15 register files in combination with a few muxes and decoders to correctly pass	**Rd (4 bits)**: The register address of the destination register (the register to write data to) **Write Data (16 bits)**: The data to write into the rd register **Cl A (1 bit)**: If enabled, register A will be cleared **Cl B (1 bit)**: If enabled, register A will be cleared **Mv A (1 bit)**: If enabled, the value in	**A (16 bits)**: The data stored in the A register **B (16 bits)**: The data stored in the B register	We will test this by writing to different registers in rd and then adding temporary outputs to the outputs of those registers to make sure that they are correctly being written to. We will also test the move functionality by first writing data to a random register and then trying to move it to A or B. Clear can then be easily tested by turning on the clear bit after the move test

	around the data.	the Mv register will be copied to A **Mv B (1 bit)**: If enabled, the value in the Mv register will be copied to B **Mv Reg (4 bits)**: The register address of the register to get data to move into A or B		
MEM	Main memory, the write data will be written to the address that is given if the write bit is a 1 if it isn't, then the memory will output the value at that address. There will be some addresses that are reserved for I/O and maybe other things that can be written to. The things that can and cannot be written to or read from will be controlled by a memory management unit. For more, see the Memory Section **To Build:** We will createthe memory	**Write Data (16 bits):** Data to be written in main memory **Address (16 bits):** Address where the data will be written **Write (1 bit):** If enabled, the write data will be written to the address **Clock (1 bit):** The clock of the processor, data will be written with the clock is enabled	**MemData (16 bits):** Data being read from main memory	We will put data in an address, then try to read from that address and see if it is the same thing.

	management unit using logic gates with some constants to determine which memory address can be accessed.			
SE	Sign extend, takes the data in and copies the most significant bit 8 times to create a 16 bit value. **To Build:** Copy the most significant bit of the input and use it to fill in the 8 most significant bits of the output.	**Data in (8 bits):** Data to be sign-extended	**Data out (16 bits):** Data after being sign-extended	To test this, we will just insert different signed 8 bit values and make sure that the correct sign-extended value is outputted
ZE	Zero extend, takes the data in and makes the 8 most significant bits of the output 0 and the 8 least significant bits are the bits from the input. **To Build:** Take in the input and then add 8 zero bits to the start.	**Data in (8 bits):** Data to be zero-extended	**Data out (16 bits):** Data after being zero extended	To test, we will pass in data and then make sure that the first 8 bits are 0 and the next 8 are the input bits
MUX	Takes in 2, 4, 8, or 16 inputs and then outputs one of them based on	**Data in (x bits):** Data inputted into the mux, could be any number of inputs (2, 4, 6, or 16) and the inputs	**Data out (x bits):** The chosen one of the inputted bits will be outputted.	To test these, we can insert different inputs and the test to make sure all of the codes make the correct

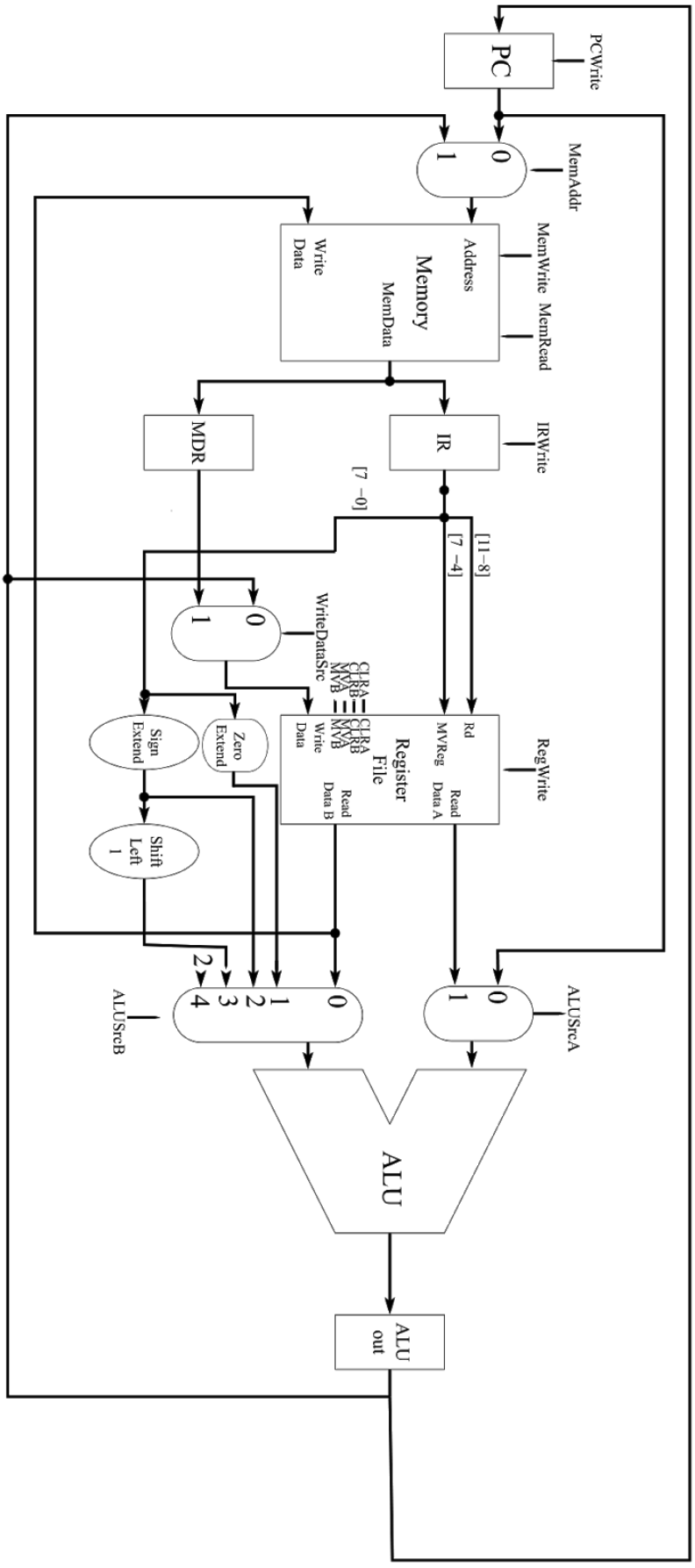
	a 1, 2, 3, or 4 bit number inputted. **To Build:** Use many smaller muxes to determine the bits that you need and then we can build on our own muxes to create large ones.	could have any amount of bits depending on the mux **Data Code (y bits):** The code that decides which of the inputs will be outputted, could be a different amount of bits depending on the mux (normally between 1 and 4)		output.
ALU	Takes in two inputs and an opcode and performs an operation on the two numbers. **To Build:** A combination of logic gates that make up one bit ALUs and then combine them into a larger 16 bit ALU	**Data in A (16 bits):** The first input into the ALU to perform operations on. **Data in B (16 bits):** The second input into the ALU to perform operations on. **Opcode (3 bits):** The code that decides which operation will be performed on the two inputs **Values:** 000: and 001: or 010: add 011: pass through a 100: pass through b 101: sll 110: subtract 111: slt	**Data Out (16 bits):** The result of the operation decided by the opcode performed on the A and B inputs. **Overflow (1 bit):** 1 if the operation overflowed, 0 if it didn't	To test the ALU, we will make two inputs and then test all of the various operations on the two numbers.
Control	Uses the op code to produce all of the control signals. **To Build:** It's a state machine	**Opcode (4 bits):** The code that determines	**Control:** All of the control signals in the table below.	To test we will run our instructions and make sure that all of the control signals are correct

	depending on what part of the instruction it's on.			

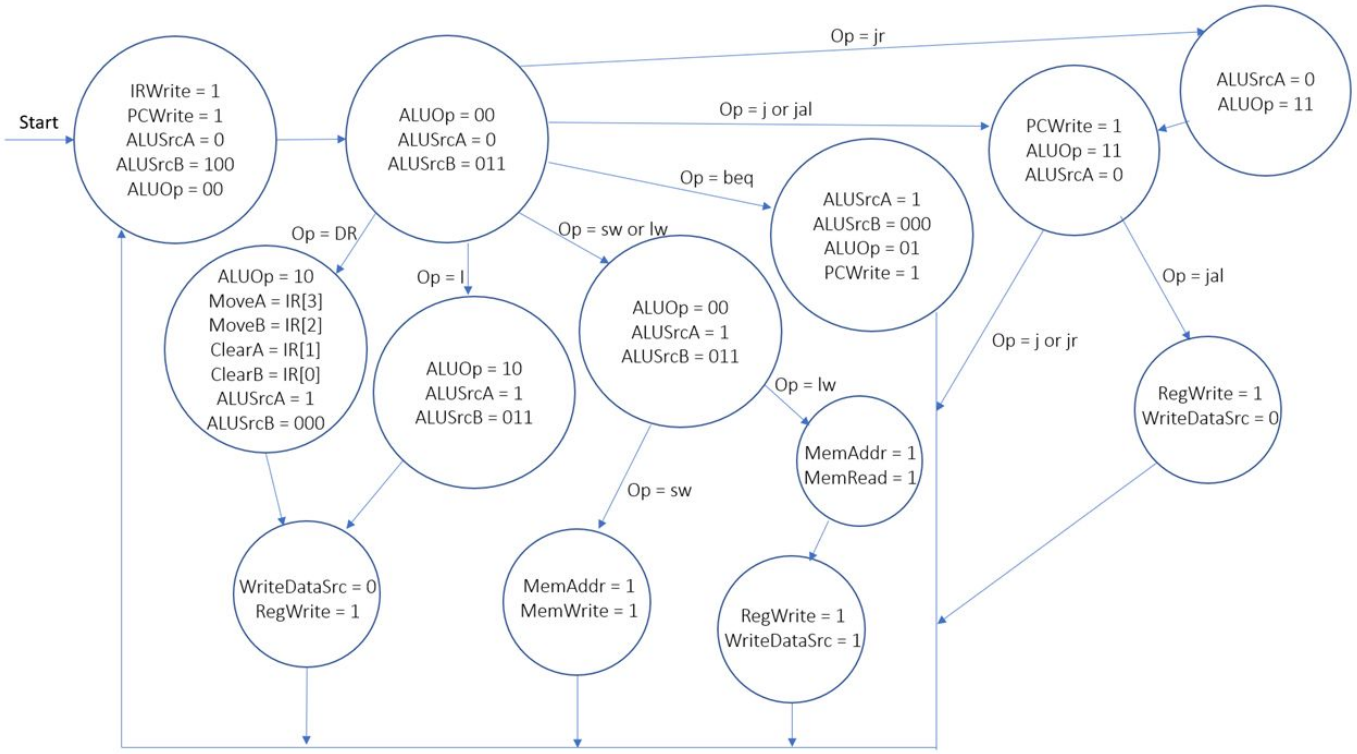
Control Signal List

Control Signal	Description
PCWrite	Determines whether or not you want to write to the pc register
ALUSrcB	Determines what value goes into the ALU: either B register, sign-extended immediate, zero-extended immediate, or sign-extended shifted immediate.
Jump	Determines if instruction is jump, jal
JumpReg	Determines if instruction is jr
Branch	Determines if instruction is beq
CLRA	Determines if A register should be cleared
CLRB	Determines if B register should be cleared
MVA	Determines if value from MVReg should be written in A register
MVB	Determines if value from MVReg should be written in B register
WriteDataSrc	Determines whether you're writing data into the register from ALUOut or MDR
IRWrite	Determines if IR register can be written to
MemAddr	Determines whether you're looking for the ALUOut or PC address in memory
MemWrite	Determines if memory will be written to
MemRead	Determines if memory will be read
RegWrite	Determines if WriteData is written into the rd register

Datapath Diagram



Finite State Machine for Control



Integration Testing

ZE, SE, Shift, ALU, ALUOut - Two numbers are input in place of the A and B registers. We will test out various scenarios where a value must be shifted, zero extended, or sign extended, then put into the ALU. We can then check the ALUOut register to make sure it is the expected value.

MEM, IR, MDR, REG - We can make sure that values can be read from memory and stored in the various registers as intended. We can also look at the output of A and B to make sure CLRA, CLRB, MVA, and MVB are functioning correctly.

PC, MUX, MEM - We can test various inputs for the jump instructions to make our datapath works correctly with jumps and the appropriate address is being jumped to.

Procedure Call Conventions

Procedure Call Conventions

Registers

- The zero register cannot change
- sp should be unchanged when returning from a procedure
- All other registers are mutable in procedures

Stack

- All mutable registers should be saved on the stack
- Extra arguments should be placed in the stack at the lowest value and increase in address
- Extra return values should be placed at the highest value addresses in the stack and count down

Example

//Registers A and B start with an arbitrary words
//proc has four inputs and three outputs (arg1, arg2, arg3, arg4, ret1, ret2, ret3)
//t0 = arg1, t1 = arg2, t2 = arg3, t3 = arg4

```
mv   $0,  $sp, 1, 0, 0, 1     #put the stack pointer in A and clear B
addi $sp, -8                  #add four word places to the stack
mv   $0,  $t0, 1, 0, 0, 0     #put arg1 into A
or   $a0, $t1, 1, 0, 0, 0     #put arg1 in a0 and arg2 in A
or   $a1, $sp, 1, 0, 0, 0     #put arg2 in a1 and put sp in A
mv   $0,  $t2, 0, 1, 0, 0     #put arg3 in B
sw   $0,  0                   #save arg3 on the stack shifted 0 words
mv   $0,  $t3, 0, 1, 0, 0     #put arg4 in B
sw   $0,  1                   #save arg4 on the stack shifted 1 word
mv   $0,  $ra, 0, 1, 0, 0     #put the return address in B
sw   $0,  2                   #save ra on the stack shifted 2 words
jal  proc                     #jump to proc
mv   $0,  $v0, 1, 0, 0, 0     #put ret1 in A
or   $t0, $sp, 1, 0, 0, 0     #put ret1 in t0 and put sp in A
lw   $t1, -1                  #load ret2 into t1
lw   $t2, -2                  #load ret3 into t2
lw   $ra, 3                   #load the return address into ra
mv   $0,  $sp, 1, 0, 0, 0     #put the stack pointer in A
addi $sp, 8                   #take four word places from stack
```

Code Fragments with Machine Code

Loading in a 16-bit integer:

```
lli $A, 0x16                        1011 1000 0001 0110
sll $A, 0x8                         1101 1000 0000 1000
ori $A, 0x21                        1100 1000 0010 0001
```

Results in the Register:

A: 0x1621

Loading in two numbers and adding them:

```
lli $A, 0x31                        1011 1000 0011 0001
lli $B, 0x02                        1011 1001 0000 0010
add $t0, $0, 0, 0, 1, 1             0000 1010 0000 0011
```

Results in the Register:

A: 0x0000
B: 0x0000
t0: 0x0033

Looping and iteration:

```
           add $t0, $t0, 0, 0, 1, 1   0000 1010 1010 0011
           lli $B, 0x05               1011 0101 0000 0101
Loop:      addi $A, 1                 1101 1000 0000 0001
           add $t0, $0, 0, 0, 0, 0    0000 1010 0000 0000
           beq $B, Loop               0111 1001 1111 1101
           add $A, $0, 0, 0, 0, 1     0000 1000 0000 0001
```

Results in the Registers:

A: 0x000A
B: 0x0000
t0: 0x0028

Euclid's Algorithm:

```
31  relPrime:
32              # n is already in $a0 from where this was called
33              lli $a1, 2              # store m in a1                         1011 0101 0000 0010
34  loop:       lli $B, 4               # load 4 into B                         1011 1001 0000 1000
35              mv $0, $sp, 1, 0, 0, 0  # move sp into A                       0101 0000 0001 1000
36              sub $sp, $a1, 0, 1, 0, 0 # decrease sp by 4 and move $a1 into $B 0001 0001 0110 0100
37              mv $0, $sp, 1, 0, 0, 0  # move the value in $sp into $A       0101 0000 0001 1000
38              sw $0, 0                # stores m on the stack                 1001 0000 0000 0000
39              mv $0, $a0, 0, 1, 0, 0  # moves n to $B                        0101 0000 0101 0100
40              sw $0, 1                # stores n on the stack                 1001 0000 0000 0100
41              mv $0, $ra, 0, 1, 0, 0  # move $ra into B                      0101 0000 0010 0100
42              sw $0, 2                # store $ra on the stack                1001 0000 0000 1000
43              jal gcd                 # jump into the gcd function            1110 0000 (address of gcd)
44              mv $0, $sp, 1, 0, 0, 1  # put sp into $A                       0101 0000 0001 1000
45              lw $a0, 1               # load n back into $a0                  0100 0101 0000 0100
46              lw $a1, 0               # load m back into $a1                  0100 0110 0000 0000
47              lw $ra, 2               # load ra back                          0100 0010 0000 1000
48              lli $A, 3               # put 4 into A                          1011 1000 0000 1000
49              add $sp, $0, 0, 0, 0, 0 # add 4 back to the stack pointer      0000 0001 0000 0000
50              mv $0, $v0, 1, 0, 0, 0  # put the result of gcd into $A        0101 0000 0111 1000
51              lli $B, 1               # put 1 into $B                         1011 1001 0000 0001
52              beq $0, INCREMENT       # if result == 1, loop                  0111 0000 0000 0010
53              mv $0, $a1, 1, 0, 0, 1  # move $a1 into A and clear B           0101 0000 0110 1001
54              add $v0, $0, 0, 0, 0, 0 # put m into $v0 to return            0000 0111 0000 0000
55              j DONE                  # if result != 1, then return m         1010 0000 (address of DONE)
56  INCREMENT:
57              add $a1, $0, 0, 0, 0, 0 # add 1 to m in $a1                   0000 0110 0000 0000
58              j LOOP                  # jump to loop                          1010 0000 (address of LOOP)
59
```

```
60  gcd:
61              mv $0, $a0, 1, 0, 0, 1    # move $a0 into A and clear B          0101 0000 0101 1001
62              beq $0, RETURNB           # if a == 0, return b                  0111 0000 0000 1010
63
64  LOOP2:      mv $0, $a1, 0, 1, 1, 0    # move $a1 into B and clear A          0101 0000 0110 1000
65              beq $0, RETURNA           # if b == 0, return a                  0111 0000 0000 1010
66              mv $0, $a0, 1, 0, 0, 0    # move $a0 back into A                 0101 0000 0101 1000
67              slt $t0, $0, 0, 0, 0, 0   # check if a < b                       0110 1010 0000 0000
68              beq $0, ELSE              # if a !< b go to the else             0111 0000 0000 0010
69              sub $a0, $0, 0, 0, 0, 0   # a = a - b                            0001 0101 0000 0000
70              j LOOP2                                                          1010 0000 (address of LOOP2)
71  ELSE:
72              mv $0, $a0, 0, 1, 0, 0    # move $a0 into B                      0101 0000 0101 0100
73              mv $0, $a1, 1, 0, 0, 0    # move $a1 into A                      0101 0000 0110 1000
74              sub $a1, $0, 0, 0, 0, 0    # b = b - a                           0001 0110 0000 0000
75              j LOOP2                                                          1010 0000 (address of LOOP2)
76
77  RETURNB:
78              mv $0, $a1, 1, 0, 0, 1    # move $a1 into A and clear B          0101 0000 0110 1001
79              j DONE                                                           1010 0000 (address of DONE)
80  RETURNA:
81              mv $0, $a0, 1, 0, 0, 1    # move $a0 into A and clear B          0101 0000 0101 1001
82  DONE:
83              mv $0, $ra, 1, 0, 0, 0    # move $ra into A
84              jr $0, $0, 0, 0, 0, 0     # jump to the return address in A      0100 0000 0000 0000
```

84 bytes required to store
When called with 0x13B0, 71,788 instructions and 235,874 cycles
Average CPI: 3.286