Violet-2223b-01

Jackson Shen and Jenna Strassburger

Table of Contents

Design Description

We will be working on a load-store architecture, similar to Risc-V. Our focus will be on ensuring the processor runs quickly, with a secondary focus on ease of programming for the user. Consistency will be maintained when possible, however the main goal of our processor will be to execute the desired program as fast as possible. In order to maximize efficiency, we are willing to work with a larger instruction set, as a trade-off for the versatility of a particular instruction.

Measurement of Performance

The key measure of performance will be the amount of time it takes for the smallest relatively prime program, implemented with Euclid's algorithm, to execute. Additionally, we will consider the raw values of the average cycles per instruction and the clock frequency.

Register Usage (16 bit registers)

Register	Name	Description	Saver
x0	zero	Zero constant	
x1	ra	Return address	Caller
x2	sp	Stack pointer	Callee
x3-x4	t0-t1	Temp registers	Caller
x5-x9	s0-s4	Saved registers	Callee
x10-x11	a0-a1	Return registers	Caller
x12-x14	a2-a4	Fn-args	Caller
x15	at	Assembler temp	Caller

Machine Language Instruction

Inst	Name	FMT	ID	Desc	Comments
add	Add	R	0000	R[rd] = R[rs1]+R[rs2]	Add rs1 and rs2, store in rd
grt	Greater than?	R	0001	R[rd] = (rs1>rs2)?1:0	Compare the argument. Set rd to 0 if rs1 <= rs2 else 1
sub	Subtract	R	0010	R[rd]=R[rs1]-R[rs2]	Subtract rs1 and rs2, store in rd
eq	Equal?	R	0011	R[rd] = (rs1==rs2)?0:1	Compare rs1 and rs2. Set rd to 0 if not equal, else 1
jalr	Jump and link register	R	0100	R[rd] = PC + 2 PC = R[rs1]	Jump the amount of the immediate bytes relative to rs1
bne	Branch not equal	R	1011	if (rs1 != rs2): PC = R[rd]	Change raw address if not equal
lui	Load upper immediate	I	0101	R[rd] = SE(imm)<<8	Load constant and shift 8
lli	Load lower immediate	I	1111	R[rd] = ZE(imm)	Load constant and zero extend
addi	Add immediate	M	1000	R[rd] = R[rs1]+SE(imm)	Add immediate to register
lw	Load word	M	1001	R[rd] = M[R[rs1] + SE(imm)]	Word from memory to register
sw	Store word	M	1010	M[R[rs1] + SE(imm)] = R[rd]	Word from register to memory
wri	Write	M	1100	If (rs1 = <special mem place>): <special mem stuff> = R[rd]	Write to output port
rea	Read	M	1101	If (rs1 = <special mem place>): R[rd] = <special mem stuff>	Read from input port

Instruction Diagrams

R-type instructions (inst rd, rs1, rs2)

rs2	rs1	rd	4 bit ID

I-type instructions (inst rd, imm)

8 bit immediate	rd	4 bit ID

M-type instructions (inst rs1, rs2, imm)

4 bit immediate	rs1	rd	4 bit ID

Addressing mode and dealing with immediates

We will be using relative addressing, where branching is relative to the current pc. We will sign-extend immediates.

Calling conventions

The caller will save registers that it will need to use in the future like return address and temporaries. All other registers will be saved by the callee. Each function keeps track of its usage of the stack. It will clear out information and return the stack pointer to its former location before it returns. Finally, a function will never modify any part of the stack above its original location.

RTL Description

R Type

add/sub	**grt/eq**	**jalr**
IR <= Mem[PC] PC <= PC +2		
A <= Reg[IR[11:8]] B <= Reg[IR[15:12]]		
add: ALUOut <= A + B *sub*: ALUOut <= A - B	ALUOut <= A - B	ALUOut <= PC+2
Reg[7:4] <= ALUOut	*grt*: Reg[7:4] = reg *eq*: Reg[7:4] = zero	Reg[IR[7:4]] <= ALUOut
		ALUOut <= A
		PC <= ALUOut

M Type

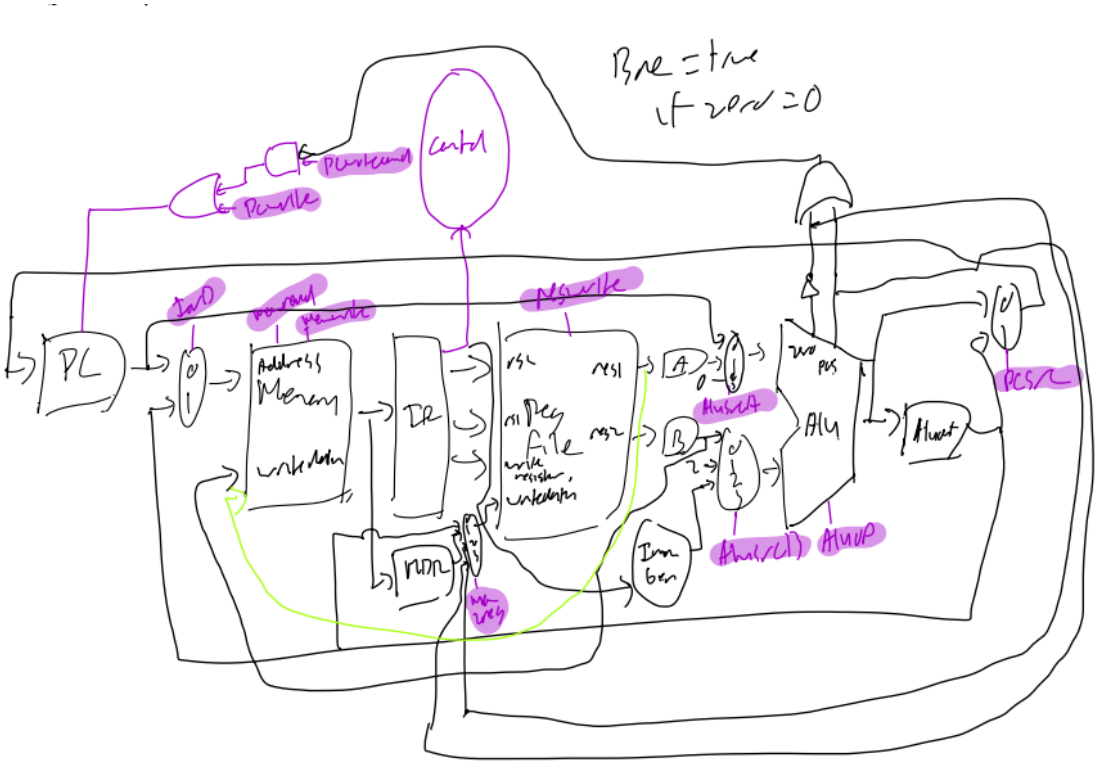
addi	**lw**	**sw**
IR <= Mem[PC] PC <= PC +2		
A <= Reg[IR[11:8]] B <= Reg[IR[15:12]]		
ALUOut <= A + SE(imm)		
Reg[IR[7:4]] = ALUOut	MDR <= Memory[ALUOut]	Memory[ALUOut] = B
	Reg[IR[7:4]] <= MDR	

I Type

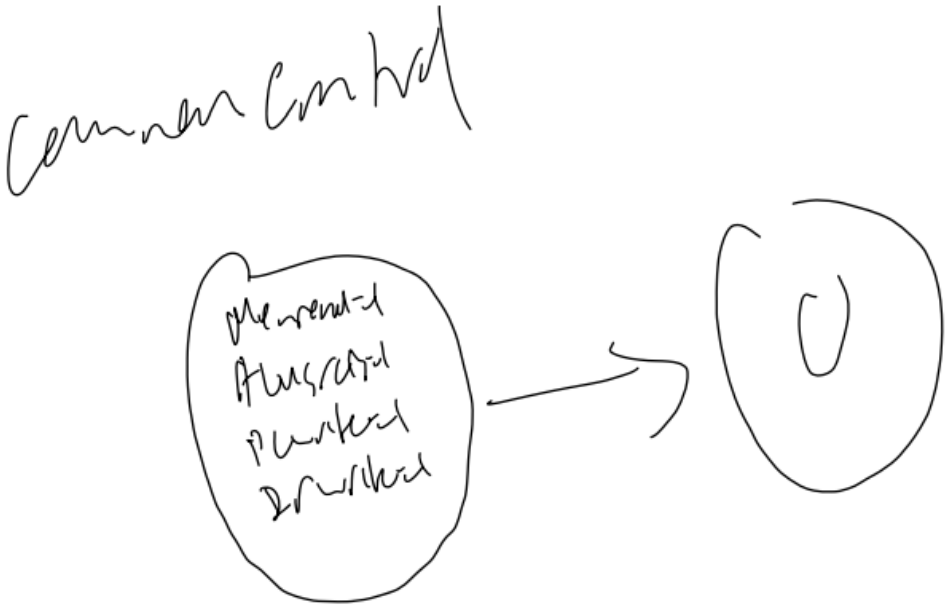
bne	**lui**	**lli**
IR <= Mem[PC] PC <= PC +2		

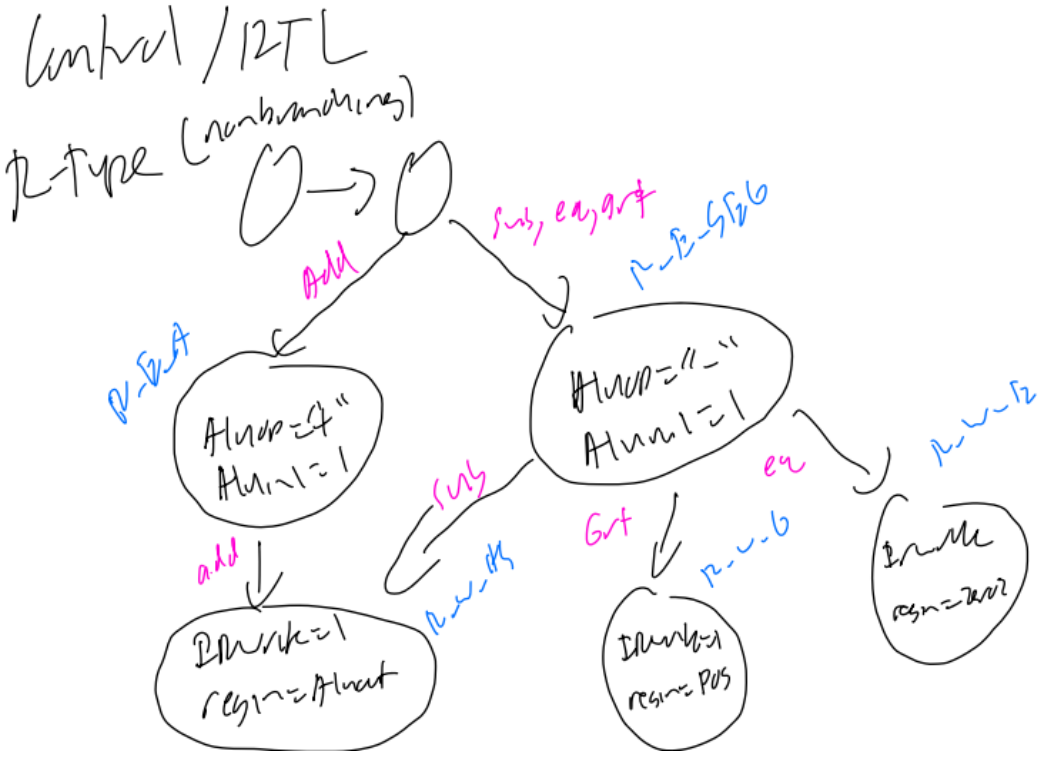
A <= Reg[IR[11:8]] B <= Reg[IR[15:12]]		
ALUout <= A - B	ALUOut = 0 + Imm << 8	ALUOut = 0+Imm
ALUout <= C + 0 If (ALUout != 0): PC <= ALUOut	Reg[IR[7:4]] <= ALUOut	

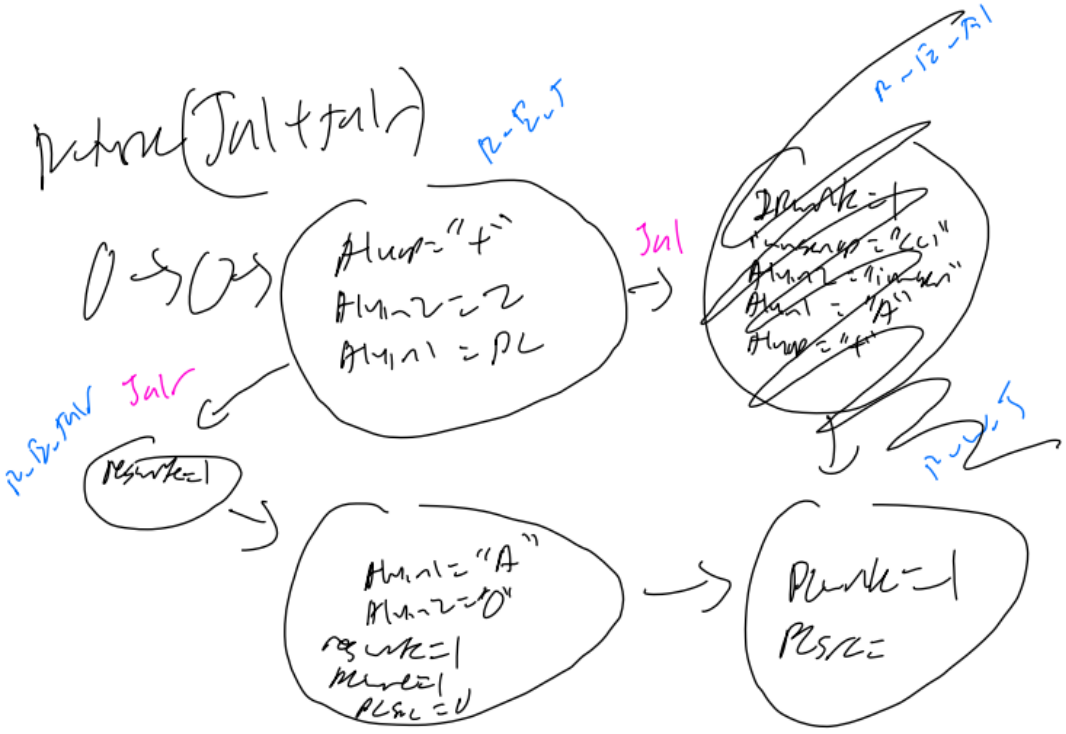
Datapath

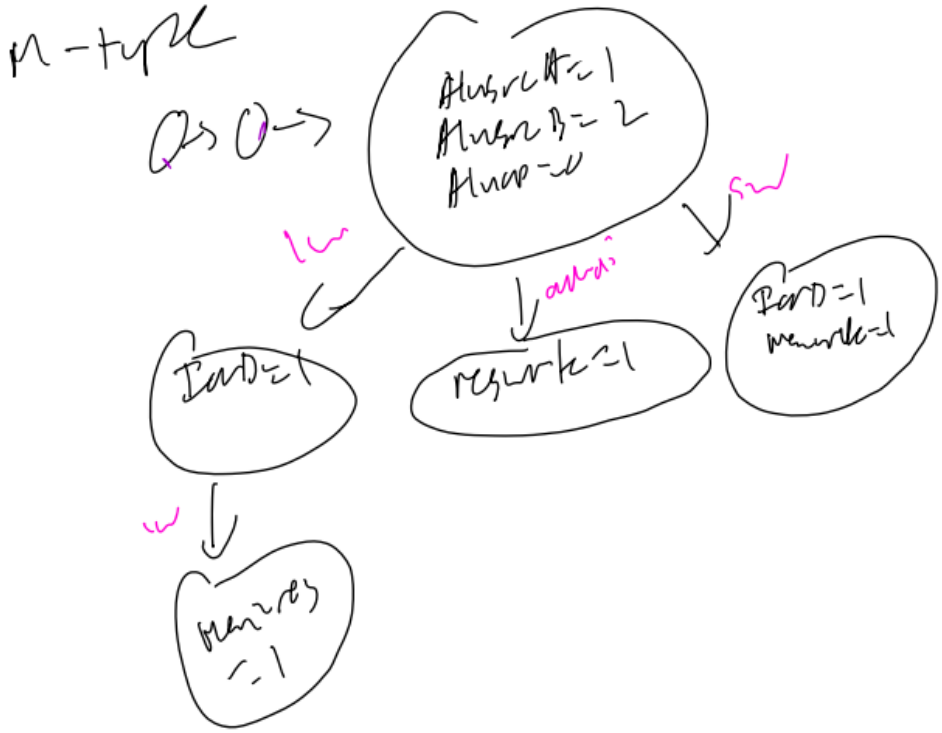


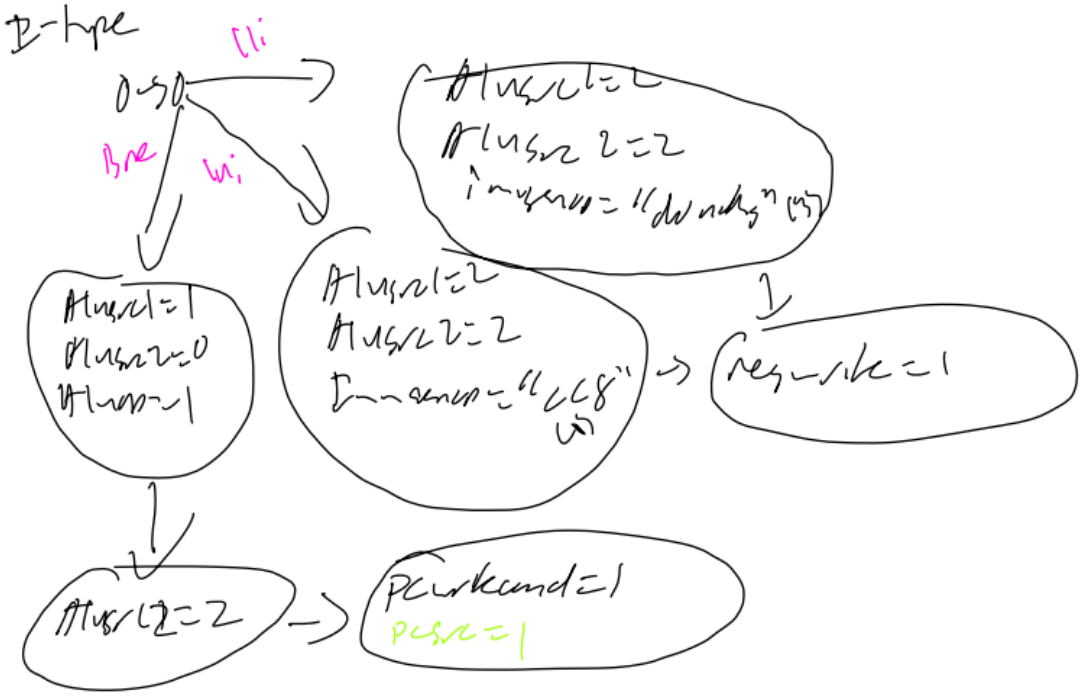
Controls











Sample RelPrime and GCD Assembly Code

Given Code

```
int relPrime(int n) {
   int m;
   m = 2;
   while (gcd(n, m) != 1) {  // n is the input from the outside world
     m = m + 1;
   }
   return m;
}
```

Given Code

```
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;
}
```

Translated to Assembly

```
lli x15, 32
lui x1, 20
add x15, x15, x1
lw x1, x15, 0 //x15 is special mem address, x1 is n (input)
lli x7, CONTINUE
lli x10, IF
lli x9, STARTWHILE
lli x14, GCD
lli x13, RWHILE
lli x11, RCONTINUE
lli x6, 1

RELPRIME: addi x2, x0, 2 // m = 2
RWHILE: jalr x5, x14, x0 // get back result in x3
```

```
bne x11, x6, x3
sw x2, x15, 0

RCONTINUE: addi x2, x2, 1
jalr x0, x13, x0

GCD: addi x3, x2, 0// return x3
addi x4, x1, 0
STARTWHILE: bne x7, x4, x0
DONE: add x15, x15, x1
jalr x0, x5, x0 //jump back to relprime
CONTINUE: grt x12, x3, x4
bne x10, x12, x0
ELSE: sub x4, x4, x3
jalr x0, x9, x0
IF: sub x3, x3, x4
jalr x0, x9, x0
```

Retro

Initial Mistakes

- Issues with completion of tests for each component
- Band-aid temporary solutions vs root cause solutions
- Lack of organization
    - Naming conventions
    - File structure
- Lack of design preparation
    - Issues with determining which components needed to be clocked
- Communication
    - Certain teammates had exaggerated contributions, leading to confusion and conflict about what needed to be done
- Procrastination
    - After our early design, we did not implement components until it was too late
    - Did not leave time to debug
    - Did not leave time for good testing practices

Fixes

- Designing from the ground up
    - Working through each scenario
    - Adhering to naming conventions
- Documentation
    - Keeping the datapath up to date
    - Maintaining comments about our RTL and Control
- Testing
    - Working through each instruction one at a time
    - More thoughtful and thorough test cases
- Division of problems
    - Breaking relprime down into smaller, easily debuggable problems
- Ability to identify when a problem is out of scope
    - Redesign to multi cycle vs pipeline
- Team meeting about communications, expectations, and group dynamic moving forward

Takeaways

- Start implementation early
- There is no such thing as too many tests
    - Make tests for components, instructions, and relPrime
- Focusing on design early can pay off as the project goes through
    - Every design issue causes a large amount of refactoring to fix

- Maintaining comments and clean file structure will speed up the development/debugging process greatly
- Set clear and reasonable team expectations for contributions and communication from the beginning
- Ask for help early on
    - No stupid questions, it’s always good to clarify