Violet-2223b-01

Ethan Norfleet, Jacob Scheibe, Jackson Shen, Jenna Strassburger

Milestone 1

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 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 be executed. In addition, 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]	3 registers operands; add
grt	Greater than?	R	0001	R[rd] = (rs1>rs2)?0:1	3 registers operands; compare the argument. Return 0 if rs1 <= rs2. Return 1 otherwise
sub	Subtract	R	0010	R[rd]=R[rs1]-R[rs2]	3 registers operands; subtract
eq	Equal?	R	0011	R[rd] = (rs1==rs2)?0:1	3 registers operands; compare the arguments. Return 0 if not equal. Return 1 otherwise
jalr	Jump and link register	R	0100	R[rd] = PC + 4 PC = R[rs1] + SE(imm)<<1	2 register operands; jump the amount of the immediate relative to rs1
lui	Load upper immediate	I	0101	R[rd] = SE(imm)<<8	2 register operands; load constant and shift 8
jal	Jump and link	I	0110	R[rd] = PC + 4 PC = PC + SE(imm)<<1	2 register operands; jump the amount of the immediate
addi	Add immediate	M	1000	R[rs1] = R[rs2]]+SE(imm)	2 register operands; add immediate to register
lw	Load word	M	1001	R[rd] = M[R[rs1] + SE(imm)]	2 register operands; word from memory to register
sw	Store word	M	1010	M[R[rs1] + SE(imm)] = R[rd]	2 register operands; word from register to memory
bne	Branch not equal	M	1011	if(rs1!=rs2 + SE(imm)) PC = PC + R[rs1]<<1	2 register operands; change raw address if not equal
wri	Write	M	1100	If (rs2 = <special mem	Handles writing to output port

				place>): R[rs1] = <special mem stuff>	
rea	Read	M	1101	If (rs1 = <special mem place>): R[rs2] = <special mem stuff>	Handles reading from input

Instruction Diagrams

R-type instructions

rs2	rs1	rd	2 bit op	func2

I-type instructions

8 bit immediate	rd	2 bit op	func2

M-type instructions

4 bit immediate	rs2	rs1	2 bit op	func2

Addressing mode and dealing with immediates

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

Calling conventions

The caller will save registers it will need to use in the future: 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. In addition, a function will never modify any part of the stack above its original location.

Sample RelPrime 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;
}
```

Translated to Assembly

```
addi sp, sp, -12
sw ra, sp, 0
sw a0, sp, 4
addi a1, x0, 2
sw a1, sp, 8

LOOP:
addi a1, a1 1
sw a1, sp, 8
jal ra, gcd
lw a1, sp, 8
Bne a0, 1, LOOP
lw a0, sp, 8
lw ra, sp, 0
addi sp, sp, 12
jalr a0, 0(ra)
```

Address	Assembly	Machine Code	Comments
0x0004	RELPRIME: addi sp, sp, -12	0100001000101000	
0x0006	sw ra, sp, 0	0000001000011010	
0x0008	sw a0, sp, 4	0100001010101010	
0x000a	addi a1, x0, 2	0010000010111000	
0x000c	sw a1, sp, 8	1000001010111010	

0x000e	LOOP: addi a1, a1 1	0001101110111000	
0x0010	sw a1, sp, 8	1000001010111010	
0x0012	jal ra, GCD	0000111000010110	
0x0014	lw a1, sp, 8	1000001010111001	
0x0016	bne a0, 1, LOOP	1110000110101011	
0x0018	lw a0, sp, 8	1000001010101001	
0x001a	lw ra, sp, 0	0000001000011001	
0x001c	addi sp, sp, 12	1100001000101000	
0x001e	jalr a0, ra, 0	0000000110100100	

Sample GCD Assembly Code

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

```
bne a0, x0, ENDIF1
jalr a1, 0(ra)
ENDIF1:
addi t0, x0, 1
LOOP:
    grt t1, a0, a1
    bne a0, x0, ENDIF2
    sub a0, a0, a1
    bne x0, t0, ENDELSE
    ENDIF2:
    sub a1, a1, a0
    ENDELSE:
    bne t0, x0, LOOP
jalr a0, 0(ra)
```

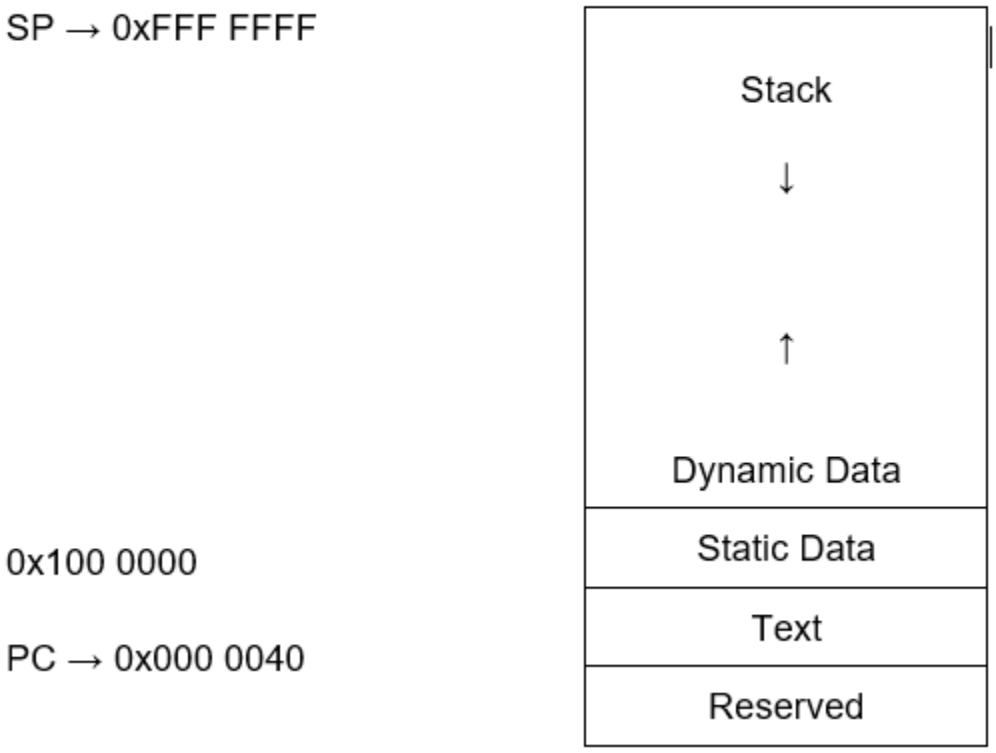
Address	Assembly	Machine Code	Comments
0x0020	GCD: bne a0, x0, ENDIF1	0100000010101011	
0x0022	jalr a1, ra, 0	0000000110110100	
0x0024	ENDIF1: addi t0, x0, 1	0001000000111000	
0x0026	LOOP: grt t1, a0, a1	1011101001000001	
0x0028	bne a0, x0, ENDIF2	0110000010101011	
0x002a	sub a0, a0, a1	1011101010100010	
0x002c	bne x0, t0, ENDELSE	0100001110101011	
0x002e	ENDIF2: sub a1, a1, a0	1010101110110010	
0x0030	ENDELSE: bne t0, x0, LOOP	0110000000111011	
0x0032	jalr a0, ra, 0	0000000110100100	

Assembly Code for Common Operations

Operation	Code
Load address	lui rd, SYMBOL[15:8] addi rd, SYMBOL[7:0]
Looping	LOOP: #do something in the loop bne t0, t1, LOOP
Conditional	bne t0, t1, ENDIF #instructions to execute if t0 == t1 ENDIF:
Reading from input port	rea, SPECIAL, t0

Writing to output port	wri t0, SPECIAL Takes whatever value is at t0 and stores it in the SPECIAL memory location

Memory Map



For now, pc starts at 0x0000