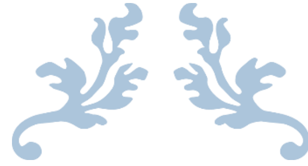


DOCUMENTATION FOR APG & ISS

Bachelor thesis

VIKRAM GANAPATHINEEDI
(EE11B048)
B VISHNU BHARGAV
(EE11B049)

Contents

Acknowledgements 2

Introduction 3

RISC-V Instruction Set Architecture 3

APG – Assembly Program Generator 4

- Program risc-v.py 4
  - Global variables: 4
  - Output files: 4
- Registers: 5
  - random_ASM_generator() 5
  - opcode_checker() 9
  - JUMP-BRANCH Instructions: 11
- Program inst2binary.c 11
  - Program execute.c: 11
- Program hexa.py 12
- Program final.sh 12

ISS - Instruction Set Simulator 13

- Utility programs 13
  - Program bindec.cpp 13
  - Program usgnbindec.cpp 14
  - Program printreg.cpp 14
  - Program printmem.cpp 14
  - Program fileprintreg.cpp 15
  - Program fileprintmem.cpp 15
  - Program str2bin.cpp 15
  - Program parseline.cpp 15
  - Program Mem.cpp 15
- Program ISS.cpp 16
- Subroutine execute.cpp 17
- Arithmetic Instructions 17
- Multiplication Instructions 19
- Memory Instructions 19
- Branch Instructions 20
  - Back Jumps 20
  - Front Jumps 21
- Header File: main.h 22
- Outputs 22

Acknowledgements

We wish to express our sincere thanks to Prof. Dr. V. Kamakoti, Project guide, for all the support and guidance. We are also grateful to Neel G (Phd scholar) and Rahul B (Research Assistant) for sharing expertise, and sincere and valuable guidance and encouragement extended to us.

*Some of the information in this report is taken from the RISC-V ISA Manual

http://riscv.org/spec/riscv-spec-v2.0.pdf.

Introduction

The purpose of this project is to develop software for testing the correctness of the microprocessor cores developed as a part of SHAKTHI PROJECT which are based on RISC-V architecture (An instruction set architecture developed in the Computer Science Division of the EECS Department at the University of California, Berkeley).

The software comprises of two main entities, ISS (Instruction Set Simulator) and APG (Assembly Program Generator).The ISS simulates the functioning of the cores .The APG generates random assembly instructions which are given to ISS and the cores, the outputs generated are compared for finding faults.

RISC-V Instruction Set Architecture

RISC-V (pronounced “risk-five”) is an instruction set architecture (ISA) that was originally designed to support computer architecture research and education, but which they now hope will become a standard open architecture for industry implementations. The goals in defining RISC-V include:

• A completely open ISA that is freely available to academia and industry.
• A real ISA suitable for direct native hardware implementation, not just simulation or binary translation.
• An ISA that avoids “over-architecting” for a particular microarchitecture style (e.g., micro-coded, in-order, decoupled, out-of-order) or implementation technology (e.g., full-custom, ASIC, FPGA), but which allows efficient implementation in any of these.
• An ISA separated into a small base integer ISA, usable by itself as a base for customized accelerators or for educational purposes, and optional standard extensions, to support general purpose software development.
• Support for the revised 2008 IEEE-754 floating-point standard [8].
• An ISA supporting extensive user-level ISA extensions and specialized variants.
• Both 32-bit and 64-bit address space variants for applications, operating system kernels, and hardware implementations.
• An ISA with support for highly-parallel multicore or many-core implementations, including heterogeneous multiprocessors.
• Optional variable-length instructions to both expand available instruction encoding space and to support an optional dense instruction encoding for improved performance, static code size, and energy efficiency.

• A fully virtualizable ISA to ease hypervisor development.
• An ISA that simplifies experiments with new supervisor-level and hypervisor-level ISA designs.

APG – Assembly Program Generator

This software is used to generate random assembly instructions in a controlled manner such that when executed by the cores, will not encounter an infinite loop or any other complications.

Language used: **Python & C**

Contains three parts:

“risc-v.py” Python program - Generates assembly code for ISS.

“inst2binary.c” C program - Converts the assembly code to binary format in accordance with RISC-V ISA.

“hexa.py” Python program – Converts the binary instructions to hexadecimal which is the input for microprocessor cores.

Program risc-v.py

```
import sys
import random
ps = 40
pagesize = ps*256
PC = 0
LJump = 0
file1 = open("output/output.s", "w")
file2 = open("output/output_readable.s", "w")
```

Global variables:

ps - Stores the value of number of pages.
pagesize - Stores the total number of instructions i.e. ps * 256(instructions per page)
PC - Program Counter
Ljump - Stores the PC of the latest jump instruction, explained later.

Output files:

output.s - Contains the assembly instructions in canonical form given to ISS.
output_readable.s - Contains assembly instructions in readable format.

Registers:

Register	Canonical Name	RISC-V Linux
x0	zero	Zero (hard-wired)
x1	ra	Return address
x2	v0	Function return value Syscall number
x3	v1	Function return value (pipe2 only)
x4	a0	Function argument register
x5	a1	Function argument register
x6	a2	Function argument register
x7	a3	Function argument register Syscall error code
x8	a4	Function argument register
x9	a5	Function argument register
x10	a6	Function argument register
x11	a7	Function argument register
x12	t0	Temporary register
x13	t1	Temporary register
x14	t2	Temporary register
x15	t3	Temporary register
x16	t4	Temporary register

Register	Canonical Name	RISC-V Linux
x17	t5	Temporary register
x18	t6	Temporary register
x19	t7	Temporary register
x20	s0	Caller-save register
x21	s1	Caller-save register
x22	s2	Caller-save register
x23	s3	Caller-save register
x24	s4	Caller-save register
x25	s5	Caller-save register
x26	s6	Caller-save register
x27	s7	Caller-save register
x28	s8 / gp	Caller-save register Global pointer
x29	s9 / fp	Caller-save register Frame pointer
x30	sp	Stack pointer (user and kernel)
x31	tp	Thread pointer (for TLS)

canregr - Dictionary which maps registers to its canonical names.

choice - It is a random number generated which is given as a seed to the function random_ASM_generator().

The program consists of two main functions:

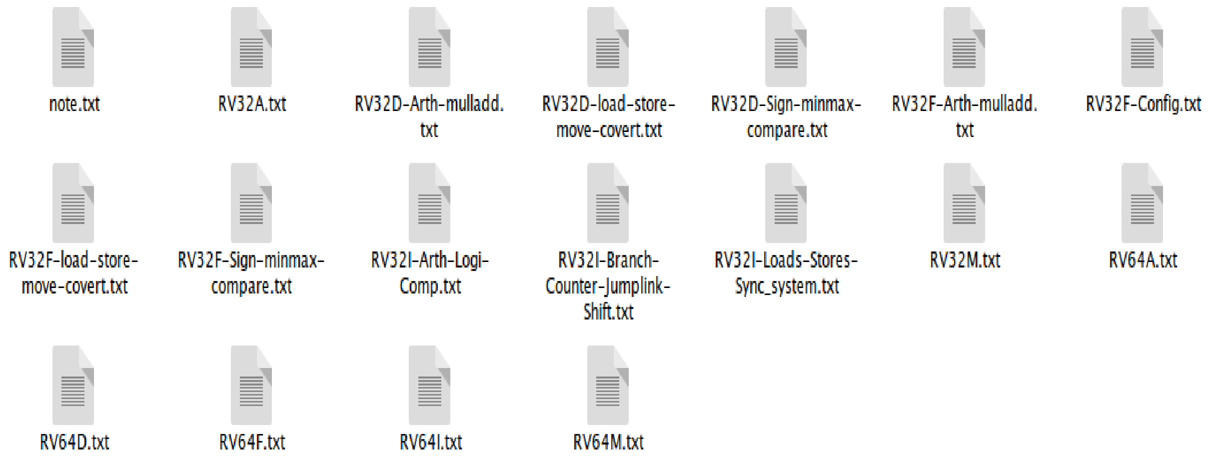
random_ASM_generator()
opcode_checker()

random_ASM_generator()

All the instructions in RISC-V ISA have been divided into 17 categories based on whether they belonged to 32I, 64I, 32F, 64F etc as mentioned in risc-v specification manual and further divided into arithmetic, branch/jump and load/store instructions as per our convenience.

Link for ISA manual: http://riscv.org/spec/riscv-spec-v2.0.pdf.

```
1  ADD
2  ADDI
3  SUB
4  LUI
5  AUIPC
6  XOR
7  XORI
8  OR
9  ORI
10 AND
11 ANDI
12 SLT
13 SLTI
14 SLTU
15 SLTIU
```



The folder catergory-grouping contains all the 17 files as shown above, a sample file RV32I-Arth-mulladd.txt is shown in the left side of the above picture.

```
list1 =[ ]
per1     = 50
list1.append(per1)
per2     = 15
list1.append(per2)
per3     = 25
list1.append(per3)
per4     = 10
list1.append(per4)
per5     = 0
list1.append(per5)
```

```
num1 = pagesize*per1/100
num2 = pagesize*per2/100
num3 = pagesize*per3/100
num4 = pagesize*per4/100
num5 = pagesize*per5/100
```

```
per = per1 + per2 + per3 + per4 + per5 + per6 + per7 + per8 + per9 + per10 + per11 + per12

if per > 100:
    sys.exit("Bye ! Next Time please provide correct inputs aggregating 100 or less")
```

There is an option to assign a percentage for each of the 17 categories which determines the composition of the output.

list1 - A list which contains the percentages of all the categories.

per1 - Stores the percentage for category 1 (explained below) which is hardcoded in the program and if needed must be changed in the program before executing it. Similarly for per2, per3 till per17.

per - Stores the total of all the percentages and ensures if it is below 100.

num1 - Number of instructions of corresponding category. Similarly for all the categories.

num - Sum of all num1, num2 etc.

If num is less than pagesize, the difference is filled by NOP instructions.

```
octyp4 = [line.strip() for line in open("./category_grouping/RV64I.txt", 'r')]
size4 = len(octyp4)
octyp5 = [line.strip() for line in open("./category_grouping/RV32M.txt", 'r')]
size5 = len(octyp5)
octyp6 = [line.strip() for line in open("./category_grouping/RV64M.txt", 'r')]
size6 = len(octyp6)
```

Octyp# - List containing all the instructions of particular category.

Size# - Contains the number of instructions in the particular category.

The category numbers are determined by the piece of code above.

The random function is seeded with the variable choice which is previously generated so that we get different set of instructions every time we run the code.

Printing to output file starts here. First R30 ad R31 are initialised.

```
Instruction = "LUI sp 00000000000000000000"
Instruction2 = "LUI r30 00000000000000000000"
file1.write("%s\n" % Instruction)
file2.write("%s\n" % Instruction2)
Instruction = "LUI tp 00000000000000000000"
Instruction2 = "LUI r31 00000000000000000000"
file1.write("%s\n" % Instruction)
file2.write("%s\n" % Instruction2)
Instruction = "ADDI tp tp 000000000101"
Instruction2 = "ADDI r31 r31 000000000101"
file1.write("%s\n" % Instruction)
file2.write("%s\n" % Instruction2)
PC = PC + 3
```

R30 and R31 registers are used only by the jump and branch instructions. They are assigned specific values, in this case R30 is 0 and R31 is 5 and are manipulated to avoid infinite loops as explained later.

```
index=1
    while(index<30):
        value=random.randint(0,1048576)
        bin_value='{0:020b}'.format(value)
        Instruction = "LUI " + canregr['r' + str(index)] +" "+bin_value
        Instruction2 = "LUI " + 'r' + str(index) +" "+bin_value
        file1.write("%s\n" % Instruction)
        file2.write("%s\n" % Instruction2)
        index=index+1
        PC=PC+1
```

Next all the remaining registers (R1 – R29) are initialised with random values. Note: R0 is always zero as specified in RISC-V ISA.

```
while (count <= (pagesize - 1 - num18)):
    i= random.randint(0,16)
    if  list1[i] != 0:
        if ((i == 0) and (num1 > 0)):
            j=random.randint(0,size1-1)
            opcode_checker(octyp1[j])
            PC = PC + 1
            num1=num1-1
            count=count+1
        elif ((i == 1) and (num2 > 0))  :
            j=random.randint(0,size2-1)
            opcode_checker(octyp2[j])
            PC = PC + 1
            num2=num2-1
            count=count+1
        elif ((i == 2) and (num3 > 0)):
            j=random.randint(0,size3-1)
            opcode_checker(octyp3[j])
            PC = PC + 1
            num3=num3-1
            count=count+1
```

Now a while loop executes until all the instructions are printed, it works as follows:

At every iteration a random integer "i" between 0 and 16 is generated, which determines the category of the instruction.

A random integer "j" is generated between 0 and size# - 1 (size# is the number of instructions in category - #). This determines the opcode, given to the opcode_checker() which prints the corresponding instruction.

PC and count are incremented, num# is decremented. If num# becomes 0, then it doesn't enter the if conditions as a result count doesn't change and the loop continues.

```
while ( num18 > 0):
    Instruction =   "ADDI zero zero 000000000000"
    Instruction2 =   "ADDI r0 r0 000000000000"
    num18 = num18 - 1
    PC = PC + 1
    file1.write("%s\n" % Instruction)
        file2.write("%s\n" % Instruction2)
return
```

The above code prints the NOP instructions at the end.

opcode_checker()

Function which prints the instructions into the output file based on the opcode received.

```
I_f         = [line.strip() for line in open("./R-I-S-U-grouping/I-f.txt", 'r')]
I_r     = [line.strip() for line in open("./R-I-S-U-grouping/I-r.txt", 'r')]
O_f         = [line.strip() for line in open("./R-I-S-U-grouping/O-f.txt", 'r')]
OI_f        = [line.strip() for line in open("./R-I-S-U-grouping/OI-f.txt", 'r')]
OI_r        = [line.strip() for line in open("./R-I-S-U-grouping/OI-r.txt", 'r')]
OI_rm_f     = [line.strip() for line in open("./R-I-S-U-grouping/OI-rm-f.txt", 'r')]
O_r         = [line.strip() for line in open("./R-I-S-U-grouping/O-r.txt", 'r')]
R4_rm_f     = [line.strip() for line in open("./R-I-S-U-grouping/R4-rm-f.txt", 'r')]
R_f         = [line.strip() for line in open("./R-I-S-U-grouping/R-f.txt", 'r')]
```

Here the instructions are divided on based on which instruction format (according to RISC-V) they belong.

31 27 26 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[12|10:5]	rs2	rs1	funct3	imm[4:1|11]	opcode	SB-type
imm[31:12]				rd	opcode	U-type
imm[20|10:1|11|19:12]				rd	opcode	UJ-type

*picture taken from RISC-v Manual

```
if opcode in R_r:
        j = random.randint(1,29)
        k = random.randint(1,29)
        l = random.randint(1,29)
        Instruction =   opcode + " "+canregr['r' + str(j)] +  ' ' + canregr['r' + str(k)] + ' '
        Instruction2 =   opcode + " "+'r'+ str(j) +  ' ' +'r' + str(k) + ' '+'r' + str(l)
        file1.write("%s\n" % Instruction)
        file2.write("%s\n" % Instruction2)
```

R_r - Contains instructions of R type and works on general purpose registers.
E.g. ADD rd, rs1, rs2

- The variables j, k, l are randomly generated between 1 and 29, and correspond to rs2, rs1 and rd.
- Instruction is a string which contains the instruction in canonical format and Instruction2 contains instruction in readable format, which are printed in corresponding output files. It is mostly similar for the other instructions, if there are any exceptions they are explained below.

R_f - Contains instructions of R type and works on floating point registers.
E.g. FMIN.S rd, rs1, rs2.

I_f - Contains instructions of I type and works on floating point registers.
E.g. FLW rd, rs1, imm.

getbits(int k) – function which returns a string of random bits of length k.
getbits function is used to get random immediate values.

U_r - Contains instructions of U type and works on general purpose registers.
E.g. AUIPC rd, imm.

Solo - Contains instructions which doesn't have any registers involved.
E.g. FENCE

O_r - Contains instructions which deal with one destination general purpose register.
E.g. RDCYCLE rd

O_f - Contains instructions which deal with one destination floating point registers.
E.g. FRCSR rd

OI_r - Contains instructions which deal with only two general purpose registers one destination and one source.
E.g. LR.W rd, rs1.

OI_f – Contains instructions which deal with only two floating point registers one destination and one source.
E.g. FMV.S.X rd, rs1.

R4_rm_f - Contains instructions which deal with four general purpose registers and rm.
E.g. FMADD.S rd, rs1, rs2, rs3, rm
rm is a 3 bit immediate.

OI_rm_f - Contains instructions which deal with one destination, one source floating point registers and rm
E.g. FSQRT.S rd, rs1, rm.

I_r - Contains instructions of type I and works on general purpose registers.
E.g. ADDI rd, rs1, imm.

For JALR instruction which is an unconditional jump, we are restricting it to forward jumps to avoid complications.
For load instructions it is ensured that memory referenced doesn't go out of bounds.

S_r - Contains instructions of type S and works on general purpose registers.
E.g. SB rs1, rs2, imm.

As in the case of load instructions, for store instructions also it is ensured memory referenced doesn't go out of bounds.

SB_r - Contains instructions of SB type and works on general purpose registers.
E.g. BNE rs1, rs2, imm.

JUMP-BRANCH Instructions:

To avoid complications certain restrictions are enforced on jump-branch instructions.

- A variable Ljump is defined, which stores a location of the last occurrence of the jump-branch instruction.
- Forward jumps are restricted for a maximum jump of 100 and backward jumps are restricted for a maximum jump of 200.
- The first back jump instruction occurs only after 216th instruction, and every successive jump branch instruction occurs after a gap of 216 instructions to avoid complications using the condition PC-Ljump > 216.
- The last front jump instruction cannot occur after the 128th instruction from the last.
- Back jump loops are always confined to 5 iterations, which is manipulated using the values of r30 and r31. The values of r30 and r31 are reset to 0 and 5 respectively once program comes out the loop using LUI and ADDI instructions. The function **branch()** is specifically defined for this purpose.

Program inst2binary.c

The purpose of this program is to convert the assembly instructions generated previously to binary format in accordance with RISC-V ISA.

The main function takes the assembly instructions file as input and parses it line by line. Each line (instruction) obtained is further parsed and stored in an array of strings variable parsline, which is given as input to execute subroutine.

Program execute.c:

```
void execute(char **parsline)
{
    char *rs1;
    char *rs2;
    char *rd;

    if(strcmp(parsline[0],"LUI") == 0)
    {
        rd = getRreg(parsline[1]);
        fprintf(output,"%.20s%s0110111\n",parsline[2],rd);
    }
    else if(strcmp(parsline[0],"AUIPC") == 0)
    {
        rd = getRreg(parsline[1]);
        fprintf(output,"%.20s%s0010111\n",parsline[2],rd);
    }
```

Every instruction has a specific set of binary format given in RISC-V ISA. The execute function checks the opcode which is the first string in the variable parsline and prints corresponding output to the output file.

Subroutine getRreg() is used to get the binary format of registers.

Program hexa.py

```
import sys
file1 = open("output/hex.s","w")

dict = {'0000': '0','0001':'1','0010':'2','0011':'3','

for line in open("output/bin.s", 'r'):

    file1.write(dict[line[0:4]])
    file1.write(dict[line[4:8]])
    file1.write(dict[line[8:12]])
    file1.write(dict[line[12:16]])
    file1.write(dict[line[16:20]])
    file1.write(dict[line[20:24]])
    file1.write(dict[line[24:28]])
    file1.write(dict[line[28:32]])
    file1.write("\n")
```

The purpose of the program is to convert binary format to hexadecimal format which is readable by the microprocessor cores.

Program final.sh

```
#!/bin/bash

python risc-v.py
gcc -o bin.o inst2binary.c
bin.o output/output.s
python hexa.py
```

The above script automates the entire process.
###
Usage:
bash final.sh
Outputs can be found in Output folder
###

ISS – Instruction Set Simulator

This software is used to simulate the functioning of microprocessors based on RISC-V ISA. It takes the assembly instructions file generated by risc-v.py program of APG, executes them and generates a register dump and memory dump as output. These output files are compared to their corresponding output files generated by the microprocessor cores to check for errors, the hexadecimal format of the assembly instructions is given as input to the cores.

Language used: **C++**

Inputs: output.s file from APG

```
int Rreg[32][64]; // Register Initializatiom
int PCbin[64];//PC in binary
bool Memo[1048576][64]; // Memory Intialization
bool outofbound[64];
long PC ,BC;
FILE *assins , *output, *memout;//file intialization
char line[70] ;
struct queue * data= (struct queue *)malloc(sizeof(struct queue));//queue intialisation
```

Rreg:	Double dimensional int array which represents the 32 general purpose registers of 64bit each.
PCbin:	Program Counter in 64bit.
Memo:	Double dimensional Boolean array which represents internal memory with 2^20 locations of 64bit each.
outofbound:	Memory for all out of bound referencing.
PC, BC:	Program counter and Branch counter.

Utility programs

Certain subroutines have been defined as utility programs for convenience as they are frequently used thought out the ISS software.

Program bindec.cpp

It contains various functions which work on binary – decimal twos-complement conversion and vice-versa as explained below.

int64_t bin2dec(int r[64])

Takes a 64 bit int array (binary input) and returns its decimal value (twos-complement).

void dec2bin(int r[64],int64_t num)

Takes a decimal value as input and stores the binary equivalent in the 64 bit int array.

void dec2bin128(char opt, int r[64], int128_t num);

Takes a decimal integer and converts into binary of 128bits and stores the lower or higher 64 bits into a 64 bit int array depending on whether the opt character variable is 'L' or 'H' respectively. By default its low.

int bin2dec32(int r[64]);

Takes a 64 bit int array (binary input) and returns decimal value of the lower 32 bits (twos-complement).

void dec2binsg32(int r[64],int num);

Takes a decimal integer and converts into binary of 32 bits, stores the sign extended 64 bits into a 64 bit int array.

Program usgnbindec.cpp

It contains various functions which work on binary - decimal unsigned conversion and vice-versa as explained below.

uint64_t ubin2dec(int r[64])

Takes a 64 bit int array (binary input) and returns its unsigned decimal value.

void udec2bin(int r[64],uint64_t num)

Takes a decimal value as input and stores the unsigned binary equivalent in the 64 bit int array.

void udec2bin128(char opt, int r[64], uint128_t num)

Takes a decimal unsigned integer and converts into binary of 64bits unsigned and stores the lower or higher 32 bits into a 32 bit register depending on the opt char variable is 'L' or 'H' respectively. By default its low.

Program printreg.cpp

void printreg(int Rindex)

Takes the index of one of the 32 general purpose registers as input and prints the 64 bit values stored in them on the terminal.

Program printmem.cpp

void printmem(uint64_t Mindex)

Takes a memory location as input and prints the 64 bit value on the terminal.

Program fileprintreg.cpp

void fileprintreg()

Prints the PC and the 64 bit values of all the 32 bit general purpose registers to the Register dump output file.

Program fileprintmem.cpp

void fileprintmem(int reg, uint64_t mloc, bool mval)*

Prints the details of the memory instructions (load – store) to the Memory dump output file.

Program str2bin.cpp

*void str2bin(int *bin,char *str)*

Copies the binary values in string str to integer array bin for variable lengths. Used for immediate values in assembly instructions.

Program parseline.cpp

*void parseline(char ** parseline,char * line)*

Takes a string variable line as input and divides the text in it, into an array of words separated by spaces and stores them in an array of strings variable parseline.

Program Mem.cpp

bool Mem(int64_t memloc)*

Takes a memory location as input and copies the value at the location into a 64 bit Boolean array if the location is between 0 and 2^{20}-1. If the location is out of bound then it always returns the 64 bit value of alternating 0 and 1 bits starting with 0 at highest significant bit.

Program ISS.cpp

This is the main program which forms the core of the Instruction Set Simulator and calls other sub routines during the execution.

```
    createqueue(data);//queue creation
    PC = 0; BC=0; // PC and BC intialization

    char *parsline[5];

    assins = fopen(argv[1],"r");    //Opens the APG output text file
    output = fopen("output/Register_dump.txt","w");
    memout = fopen("output/MemoryInst_dump.txt", "w");
    fprintf(memout, "Type\tReg\t\tMemloc\t\t\t\tValue\t\t\t\tPC\n" );
```

createqueue(): Function which creates a queue, which is used to keep track of last 200 instructions executed. This is used in execution of jump-branch instructions which is explained later.

parsline[]: Array of strings variable to store the parsed instruction.

assins: File pointer for the assembly instructions input file.

output,memout: File pointers for register-dump and memory-dump output files respectively.

```
  while(fgets(line, sizeof(line), assins) != NULL)    //Reads the assem
  {

          PC ++ ; //Program Counter Incrementation
          addq(line,PC,data);
          if(PC > 200)
          {
              delq(data);
          }
          parseline(parsline,line);
          int i=0;
          printf("PC = %ld ,Queue head : %ld ,Queue tail : %ld , %ld\n"
          execute(parsline);      //Calls the function execute() in pro
  }
```

Input instructions are fetched line by line from the file using fgets(), and stored in the string variable line.

addq(): Adds the current instruction to the queue.

delq(): Deletes the first instruction from the queue if more than 200 instructions have been executed.

The parsed line is given as input to execute() function.

Subroutine execute.cpp

```
void execute(char **parsline)
{
    int rd,rs1,rs2;
    if(strcmp(parsline[0],"LUI") == 0)
    {
        int imm[20];
        str2bin(imm,parsline[2]);
        rd = getRreg(parsline[1]);
        lui(rd,imm);
        fileprintreg();
    }
```

Basically this function checks the opcode which is the first string in parsline variable and generates the corresponding parameters required for the execution of the instruction and calls the subroutine specific to the opcode.

Example: As in the case above lui instruction needs a 20 bit immediate and the location of the destination register, which are stored in variables rd and imm.

It is similar for all the instructions, destination register location is stored in rd and locations of the source registers are stored in rs1 and rs2 and any immediate values are stored in integer arrays of required length.

Arithmetic Instructions

ADDI adds the sign-extended 12-bit immediate to register rs1. Arithmetic overow is ignored and the result is simply the low 32-bits of the result. ADDI rd, rs1, 0 is used to implement the MV rd, rs1 assembler pseudo-instruction.

SLTI (set less than immediate) places the value 1 in register rd if register rs1 is less than the sign-extended immediate when both are treated as signed numbers, else 0 is written to rd. SLTIU is similar but compares the values as unsigned numbers (i.e., the immediate is first sign-extended to 32-bits then treated as an unsigned number). Note, SLTIU rd, rs1, 1 sets rd to 1 if rs1 equals zero, otherwise sets rd to 0 (assembler pseudo-op SEQZ rd, rs).

SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits).

ANDI, ORI, XORI are logical operations that perform bitwise AND, OR, and XOR on register rs1and the sign-extended 12-bit immediate and place the result in rd. Note, XORI rd, rs1, -1 performs a bitwise logical inversion of register rs1 (assembler pseudo-instruction NOT rd, rs).

LUI (load upper immediate) is used to build 32-bit constants and uses the U-type format. LUI places the U-immediate value in the top 20 bits of the destination register rd, filling in the lowest12 bits with zeros. The 32-bit result is sign-extended to 64 bits.

AUIPC (add upper immediate to pc) is used to build pc-relative addresses and uses the U-type format. AUIPC sign-extends the 20-bit U-immediate to 64 bits, adds it to the high 52 bits of the pc, clears the low 12 bits, then places the result in register rd.

ADD and SUB perform addition and subtraction respectively. Over flows are ignored and the low 32 bits of results are written to the destination. SLT and SLTU perform signed and unsigned compares respectively, writing 1 to rd if rs1 < rs2, 0 otherwise. Note, SLTU rd, x0, rs2 sets rd to 1 if rs2 is not equal to zero, otherwise sets rd to zero (assembler psuedo-op SNEZ rd, rs). AND, OR, and XOR perform bitwise logical operations.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in the lower 5 bits of register rs2.

The NOP instruction does not change any user-visible state, except for advancing the pc. NOP is encoded as ADDI x0, x0, 0.

ADDIW is an RV64I-only instruction that adds the sign-extended 12-bit immediate to register rs1 and produces the proper sign-extension of a 32-bit result in rd. Overflows are ignored and the result is the low 32 bits of the result sign-extended to 64 bits. Note, ADDIW rd, rs1, 0 writes the sign-extension of the lower 32 bits of register rs1 into register rd.

SLLI is a logical left shift (zeros are shifted into the lower bits); SRLI is a logical right shift (zeros are shifted into the upper bits); and SRAI is an arithmetic right shift (the original sign bit is copied into the vacated upper bits). SLLIW, SRLIW, and SRAIW are RV64I-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results.

ADDW and SUBW are RV64I-only instructions that are defined analogously to ADD and SUB but operate on 32-bit values and produce signed 32-bit results. Overflows are ignored, and the low 32-bits of the result is sign-extended to 64-bits and written to the destination register.

SLL, SRL, and SRA perform logical left, logical right, and arithmetic right shifts on the value in register rs1 by the shift amount held in register rs2. In RV64I, only the low 6 bits of rs2 are considered for the shift amount.

SLLW, SRLW, and SRAW are RV64I-only instructions that are analogously defined but operate on 32-bit values and produce signed 32-bit results. The shift amount is given by rs2.

Multiplication Instructions

MUL performs an 64-bit x 64-bit multiplication and places the lower 64 bits in the destination register. MULH, MULHU, and MULHSU perform the same multiplication but return the upper 64 bits of the full 2x64-bit product, for signed x signed, unsigned x unsigned, and signed x unsigned multiplication respectively. If both the high and low bits of the same product are required, then the recommended code sequence is: MULH[[S]U] rdh, rs1, rs2; MUL rdl, rs1,rs2.

MULW multiplies the lower 32 bits of the source registers, placing the sign-extension of the lower 32 bits of the result into the destination register. MUL can be used to obtain the upper 32 bits of the 64-bit product, but signed arguments must be proper 32-bit signed values, whereas unsigned arguments must have their upper 32 bits clear.

Memory Instructions

Load and store instructions transfer a value between the registers and memory. Loads are encoded in the I-type format and stores are S-type. The effective byte address is obtained by adding register rs1 to the sign-extended 12-bit offset. Loads copy a value from memory to register rd. Stores copy the value in register rs2 to memory.

The LD instruction loads a 64-bit value from memory into register rd for RV64I. The LW instruction loads a 32-bit value from memory and sign-extends this to 64 bits before storing it in register rd for RV64I. The LWU instruction, on the other hand, zero-extends the 32-bit value from memory for RV64I. LH and LHU are defined analogously for 16-bit values, as are LB and LBU for 8-bit values. The SD, SW, SH, and SB instructions store 64-bit, 32-bit, 16-bit, and 8-bit values from the low bits of register rs2 to memory.

Branch Instructions

All branch instructions use the SB-type instruction format. The 12-bit B-immediate encodes signed offsets in multiples of 2, and is added to the current pc to give the target address.

Branch instructions compare two registers. BEQ and BNE take the branch if registers rs1 and rs2 are equal or unequal respectively. BLT and BLTU take the branch if rs1 is less than rs2, using signed and unsigned comparison respectively. BGE and BGEU take the branch if rs1 is greater than or equal to rs2, using signed and unsigned comparison respectively. Note, BGT, BGTU, BLE, and BLEU can be synthesized by reversing the operands to BLT, BLTU, BGE, and BGEU, respectively.

The jump and link (JAL) instruction uses the UJ-type format, where the J-immediate encodes a signed offset in multiples of 2 bytes. The offset is sign-extended and added to the pc to form the jump target address. Jumps can therefore target a _1 MiB range. JAL stores the address of the instruction following the jump (pc+4) into register rd. The standard software calling convention uses x1 as the return address register.

The indirect jump instruction JALR (jump and link register) uses the I-type encoding. The target address is obtained by adding the 12-bit signed I-immediate to the register rs1, then setting the least-significant bit of the result to zero. The address of the instruction following the jump (pc+4) is written to register rd. Register x0 can be used as the destination if the result is not required.

To avoid complications as also mentioned earlier BEQ, BGE, BGEU, BLT, BLTU are programmed only to do back jumps and JAL, BNE, JALR are programmed only to jump forward.

Back Jumps

```
struct node * temp = (struct node *)malloc(sizeof(node));
temp =getline((long)200+decre-1,data);
while(i <= decre*-1)
{
    PC++;
    temp = temp->next;
    char *parsline[5];

    strcpy(line,temp->line);
    //printf("%s",line );
    printf("PC = %ld ,Queue head : %ld ,Queue tail : %ld , %ld\n",PC ,da
    parseline(parsline,line);
    execute(parsline);
    //fileprintreg();
    i++;
}
```

The above piece of code is used to simulate back jumps."decre" variable contains the number of instructions to back jump. As you remember the queue contains previous 200 executed instructions, a node pointer is defined and is pointed to desired instruction to which we want to jump in the queue using getline() function.

getline():

```
struct node * getline(long z ,struct queue * data){
    struct node *temp = data ->head ;
    long i=1;
    //printf("HI hello %ld\n",z);
    while(i<z){
        temp = temp->next;
        i++;
    }
    return temp;
}
```

This function returns the node pointer to the zth instruction in the queue. So if we need a back jump of value x we need to give the parameter z as 200-x.

Front Jumps

```
while(i<inc)
{
    fgets(line,sizeof(line),assins);
    addq(line,PC,data);
    if(PC > 200)
        {
            delq(data);
        }
    i++;
}
```

The above piece of code is used to simulate the front jumps. It basically skips the instructions and moves to the desired instruction based on the value of forward jump and takes care of queue structure as well.

Header File: main.h

To make addition of new instructions easy, we have designed a modular code with lots of subroutines. So there is need of header file which contains all the declarations of the functions and include statements of subroutines.

The subroutines belonging to Utility programs, Arithmetic Instructions, Branch Instructions etc. are provided with separate folders.

To add a new instruction to the ISS one must follow the following steps:

- Create a subroutine for the instruction and place it in the appropriate folder.
- Add a new if condition in execute.cpp taking care of required parameters.
- Update main.h with declarations and include statements.

Outputs

Two output files, register dump and memory dump are generated in output folder which are used to test the micro-processor cores.