RISC-V Programming & Design Labs

Smartcomputerlab

In this document we provide the preparations for several laboratories (labs) dealing with RISC-V programming and design. The user should have some practical knowledge of C programming language as well as Verilog design language.

Programming

The programming part (two laboratories) is prepared to experiment with RISC-V ISA and corresponding assembly language.
We will use C language to generate the assembly code. The generated code will be use to experiment with assembly language in order to understand the ISA and the essential assembly level instructions.

Design

The design part is related to the limited implementation of some of ISA instructions (R-type) with Verilog language that allows us to model and simulate the design at the architectural – register transfer (RT) level.
To follow this laboratory the user have to study the essential features of RISC-V architecture and its implementation.

The first two laboratories deal with assembly level programming, the third one introduces a limited Verilog model for the implementation of R-type instructions and a single step execution of these instructions.

The programming labs may be carried locally on the provided SBC boards :

Lichee Dock with D1 (C906) SoC or StarFive (VisionFive2)

or remotely on

LicheePi 4A with the provided **IP address for `ssh` connection**.

The design part can be done locally on any equipment with Linux (Debian/Ubuntu) or on VisionFive2 (Debian) using **`iverilog`** compiler/simulator and **`gtkwave`** graphic window tool.

The **remote design is limited only to the textual tools** – editing/compilation/simulation with textual monitoring of the results..

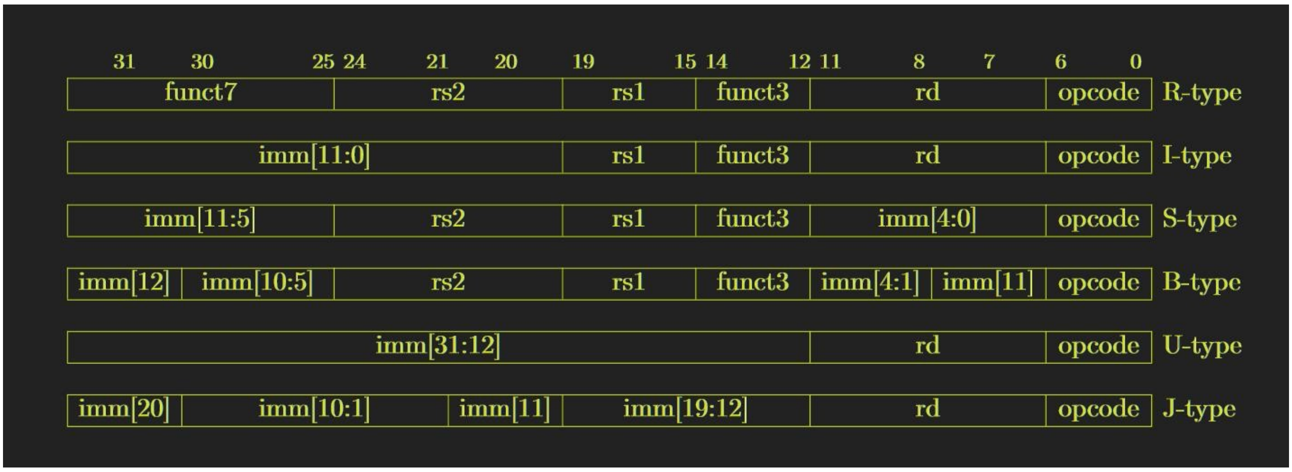
Lab 1

Assembly programming – basics

Let us look at the following tables that show the registers and the essentials formats of the assembly level instructions of RISC-V.

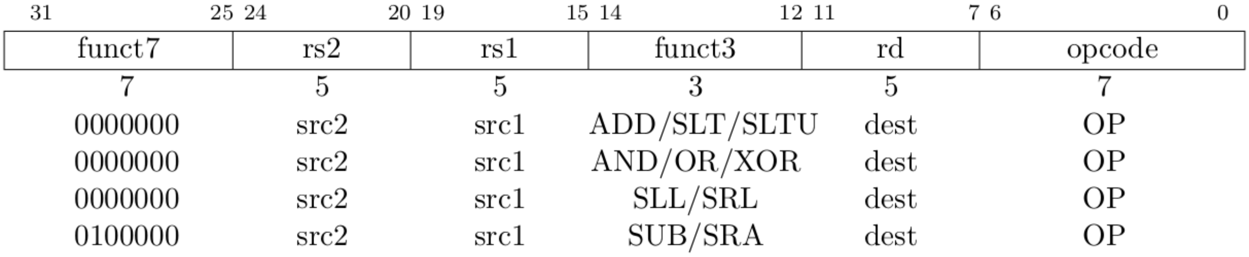
Register	ABI Name	Description	Saver
`x0`	`zero`	`Hard-wired zero`	–
`x1`	`ra`	`Return address`	`Caller`
`x2`	`sp`	`Stack pointer`	`Callee`
`x3`	`gp`	`Global pointer`	–
`x4`	`tp`	`Thread pointer`	–
`x5`	`t0`	`Temporary/alternate link register`	`Caller`
`x6-7`	`t1-2`	`Temporaries`	`Caller`
`x8`	`s0/fp`	`Saved register/frame pointer`	`Callee`
`x9`	`s1`	`Saved register`	`Callee`
`x10-11`	`a0-1`	`Function arguments/return values`	`Caller`
`x12-17`	`a2-7`	`Function arguments`	`Caller`
`x18-27`	`s2-11`	`Saved registers`	`Callee`
`x28-31`	`t3-6`	`Temporaries`	`Caller`
`f0-7`	`ft0-7`	`FP temporaries`	`Caller`
`f8-9`	`fs0-1`	`FP saved registers`	`Callee`
`f10-11`	`fa0-1`	`FP arguments/return values`	`Caller`
`f12-17`	`fa2-7`	`FP arguments`	`Caller`
`f18-27`	`fs2-11`	`FP saved registers`	`Callee`
`f28-31`	`ft8-11`	`FP temporaries`	`Caller`

Fig 1.1 Basic registers for **RVI** and **RVF**



31	30 25	24 21	20	19 15	14 12	11 8	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]	imm[10:5]	rs2		rs1	funct3	imm[4:1]	imm[11]	opcode	B-type
imm[31:12]						rd		opcode	U-type
imm[20]	imm[10:1]		imm[11]	imm[19:12]		rd		opcode	J-type

Fig 1.2 RISC-V instruction types



31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Fig 1.2a R-type instructions: two source registers (**`rs1,rs2`**) and one destination register (**`rd`**)

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
I-immediate[11:0]	src	ADDI/SLTI[U]	dest	OP-IMM
I-immediate[11:0]	src	ANDI/ORI/XORI	dest	OP-IMM

Fig 1.2b I-type instructions: one source register (**`rs1,rs2`**), one immediate value (**`imm[11:0]`** and one destination register (**`rd`**)

31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	width	dest	LOAD

31 25	24 20	19 15	14 12	11 7	6 0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode
7	5	5	3	5	7
offset[11:5]	src	base	width	offset[4:0]	STORE

Fig 1.2c S-type (**STORE**) instruction: one source register address (**`rs2`**), one base address register (**`rs1`**), and a **12-bit immediate** **`{offset[11:5],offset[4:0]}`** value.

31	30 25	24 20	19 15	14 12	11 8	7	6 0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]	opcode
1	6	5	5	3	4	1	7
offset[12,10:5]		src2	src1	BEQ/BNE	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BLT[U]	offset[11,4:1]		BRANCH
offset[12,10:5]		src2	src1	BGE[U]	offset[11,4:1]		BRANCH

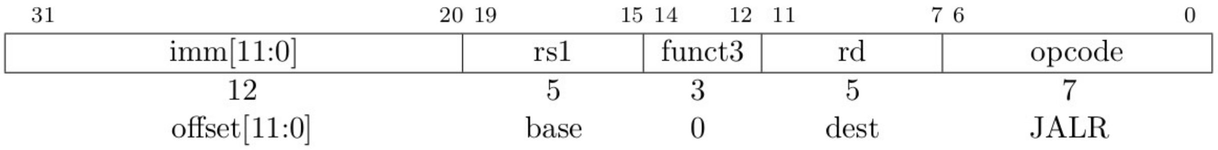
Fig 1.2d B-type (**BEQ**) instructions: takes branch if rs1 and r2 are equal, 12-bit offset is built from: **`{imm[12],imm[10:5],imm[11],imm[4:1]}`**

31 12	11 7	6 0
imm[31:12]	rd	opcode
20	5	7
U-immediate[31:12]	dest	LUI
U-immediate[31:12]	dest	AUIPC

Fig 1.2e U-type (**LUI** – load upper immediate) instruction: destination register address in (rd), immediate top 20-bit value for 32-bit data with the lowest 12 bits with zeros.

31	30 21	20	19 12	11 7	6 0
imm[20]	imm[10:1]	imm[11]	imm[19:12]	rd	opcode
1	10	1	8	5	7
	offset[20:1]			dest	JAL

Fig 1.2f J-type (**JAL**)**: unconditional jump (assembler pseudo – `j`) is encoded with `rd=x0`**

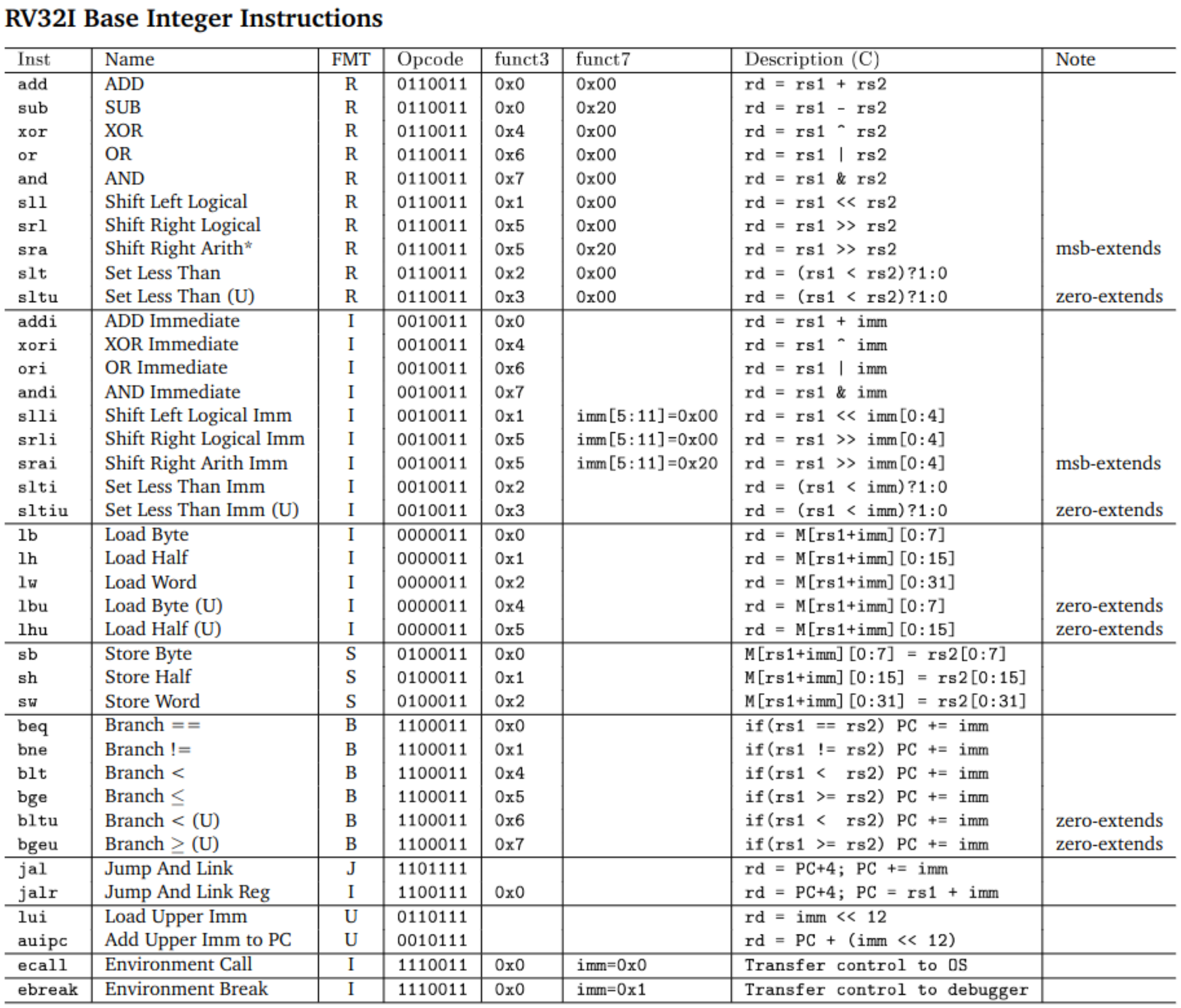


31 20	19 15	14 12	11 7	6 0
imm[11:0]	rs1	funct3	rd	opcode
12	5	3	5	7
offset[11:0]	base	0	dest	JALR

Fig 1.2f JALR uses I type encoding: the target address is obtained by adding the 12bit signed I-immediate to the register rs1; then setting the lest-significant bit to zero.

Our RISC-V processors implement **RV64G ISA** with all integer and floating point arithmetic operations, including multiplication and division.

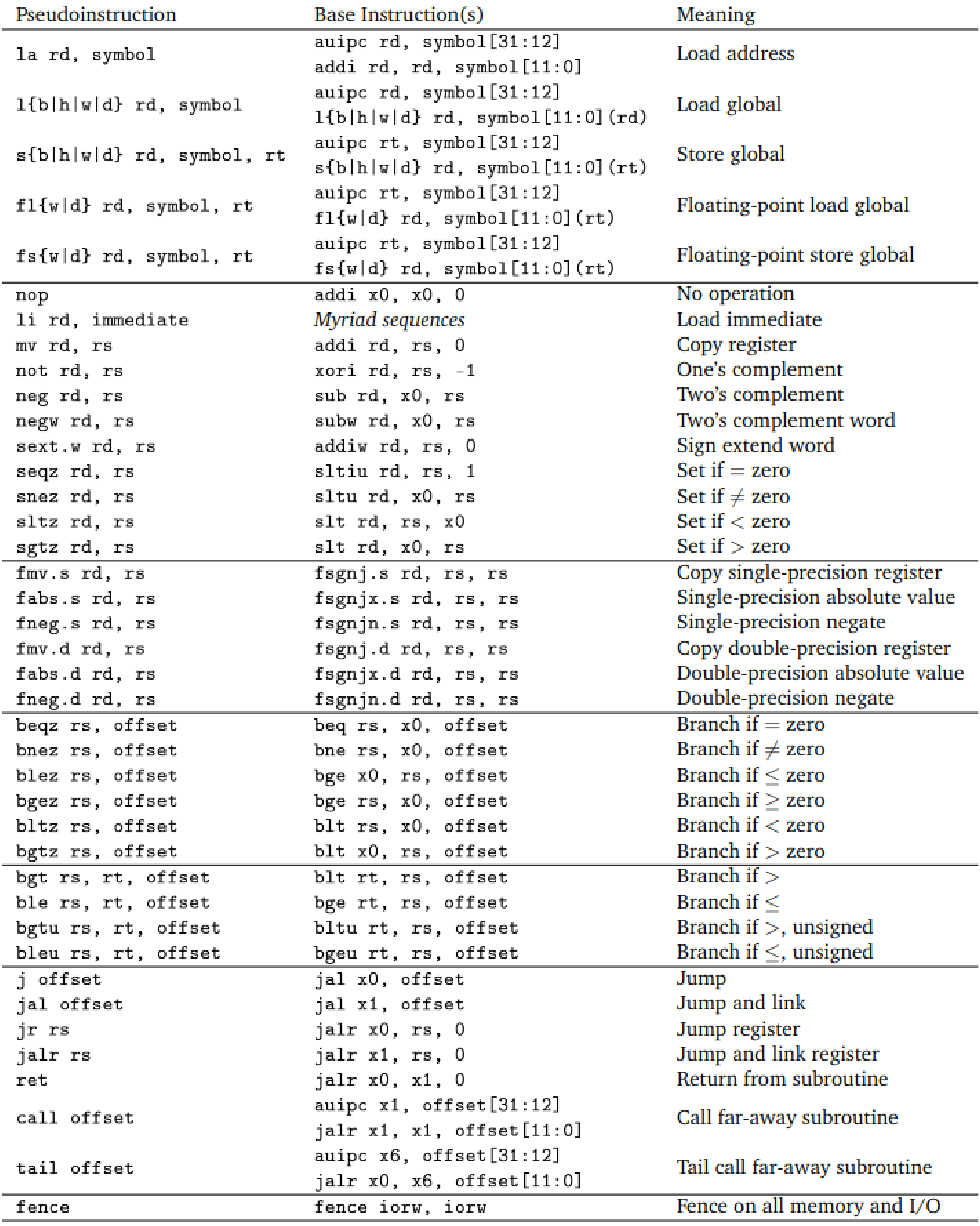
Below you will find the complete reference to RV32I **base integer** instructions.



RV32I Base Integer Instructions

Inst	Name	FMT	Opcode	funct3	funct7	Description (C)	Note
add	ADD	R	0110011	0x0	0x00	rd = rs1 + rs2	
sub	SUB	R	0110011	0x0	0x20	rd = rs1 - rs2	
xor	XOR	R	0110011	0x4	0x00	rd = rs1 ˆ rs2	
or	OR	R	0110011	0x6	0x00	rd = rs1 | rs2	
and	AND	R	0110011	0x7	0x00	rd = rs1 & rs2	
sll	Shift Left Logical	R	0110011	0x1	0x00	rd = rs1 << rs2	
srl	Shift Right Logical	R	0110011	0x5	0x00	rd = rs1 >> rs2	
sra	Shift Right Arith*	R	0110011	0x5	0x20	rd = rs1 >> rs2	msb-extends
slt	Set Less Than	R	0110011	0x2	0x00	rd = (rs1 < rs2)?1:0	
sltu	Set Less Than (U)	R	0110011	0x3	0x00	rd = (rs1 < rs2)?1:0	zero-extends
addi	ADD Immediate	I	0010011	0x0		rd = rs1 + imm	
xori	XOR Immediate	I	0010011	0x4		rd = rs1 ˆ imm	
ori	OR Immediate	I	0010011	0x6		rd = rs1 | imm	
andi	AND Immediate	I	0010011	0x7		rd = rs1 & imm	
slli	Shift Left Logical Imm	I	0010011	0x1	imm[5:11]=0x00	rd = rs1 << imm[0:4]	
srli	Shift Right Logical Imm	I	0010011	0x5	imm[5:11]=0x00	rd = rs1 >> imm[0:4]	
srai	Shift Right Arith Imm	I	0010011	0x5	imm[5:11]=0x20	rd = rs1 >> imm[0:4]	msb-extends
slti	Set Less Than Imm	I	0010011	0x2		rd = (rs1 < imm)?1:0	
sltiu	Set Less Than Imm (U)	I	0010011	0x3		rd = (rs1 < imm)?1:0	zero-extends
lb	Load Byte	I	0000011	0x0		rd = M[rs1+imm][0:7]	
lh	Load Half	I	0000011	0x1		rd = M[rs1+imm][0:15]	
lw	Load Word	I	0000011	0x2		rd = M[rs1+imm][0:31]	
lbu	Load Byte (U)	I	0000011	0x4		rd = M[rs1+imm][0:7]	zero-extends
lhu	Load Half (U)	I	0000011	0x5		rd = M[rs1+imm][0:15]	zero-extends
sb	Store Byte	S	0100011	0x0		M[rs1+imm][0:7] = rs2[0:7]	
sh	Store Half	S	0100011	0x1		M[rs1+imm][0:15] = rs2[0:15]	
sw	Store Word	S	0100011	0x2		M[rs1+imm][0:31] = rs2[0:31]	
beq	Branch ==	B	1100011	0x0		if(rs1 == rs2) PC += imm	
bne	Branch !=	B	1100011	0x1		if(rs1 != rs2) PC += imm	
blt	Branch <	B	1100011	0x4		if(rs1 < rs2) PC += imm	
bge	Branch ≤	B	1100011	0x5		if(rs1 >= rs2) PC += imm	
bltu	Branch < (U)	B	1100011	0x6		if(rs1 < rs2) PC += imm	zero-extends
bgeu	Branch ≥ (U)	B	1100011	0x7		if(rs1 >= rs2) PC += imm	zero-extends
jal	Jump And Link	J	1101111			rd = PC+4; PC += imm	
jalr	Jump And Link Reg	I	1100111	0x0		rd = PC+4; PC = rs1 + imm	
lui	Load Upper Imm	U	0110111			rd = imm << 12	
auipc	Add Upper Imm to PC	U	0010111			rd = PC + (imm << 12)	
ecall	Environment Call	I	1110011	0x0	imm=0x0	Transfer control to OS	
ebreak	Environment Break	I	1110011	0x0	imm=0x1	Transfer control to debugger	

Fig 1.3. RISC-V RV32I base instructions



Pseudoinstruction	Base Instruction(s)	Meaning

`la rd, symbol`	`auipc rd, symbol[31:12]`
	`addi rd, rd, symbol[11:0]`
		Load address

`l{b|h|w|d} rd, symbol`	`auipc rd, symbol[31:12]`
	`l{b|h|w|d} rd, symbol[11:0](rd)`
		Load global

`s{b|h|w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`s{b|h|w|d} rd, symbol[11:0](rt)`
		Store global

`fl{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fl{w|d} rd, symbol[11:0](rt)`
		Floating-point load global

`fs{w|d} rd, symbol, rt`	`auipc rt, symbol[31:12]`
	`fs{w|d} rd, symbol[11:0](rt)`
		Floating-point store global

`nop`	`addi x0, x0, 0`	No operation
`li rd, immediate`	*Myriad sequences*	Load immediate
`mv rd, rs`	`addi rd, rs, 0`	Copy register
`not rd, rs`	`xori rd, rs, -1`	One's complement
`neg rd, rs`	`sub rd, x0, rs`	Two's complement
`negw rd, rs`	`subw rd, x0, rs`	Two's complement word
`sext.w rd, rs`	`addiw rd, rs, 0`	Sign extend word
`seqz rd, rs`	`sltiu rd, rs, 1`	Set if = zero
`snez rd, rs`	`sltu rd, x0, rs`	Set if ≠ zero
`sltz rd, rs`	`slt rd, rs, x0`	Set if < zero
`sgtz rd, rs`	`slt rd, x0, rs`	Set if > zero
`fmv.s rd, rs`	`fsgnj.s rd, rs, rs`	Copy single-precision register
`fabs.s rd, rs`	`fsgnjx.s rd, rs, rs`	Single-precision absolute value
`fneg.s rd, rs`	`fsgnjn.s rd, rs, rs`	Single-precision negate
`fmv.d rd, rs`	`fsgnj.d rd, rs, rs`	Copy double-precision register
`fabs.d rd, rs`	`fsgnjx.d rd, rs, rs`	Double-precision absolute value
`fneg.d rd, rs`	`fsgnjn.d rd, rs, rs`	Double-precision negate
`beqz rs, offset`	`beq rs, x0, offset`	Branch if = zero
`bnez rs, offset`	`bne rs, x0, offset`	Branch if ≠ zero
`blez rs, offset`	`bge x0, rs, offset`	Branch if ≤ zero
`bgez rs, offset`	`bge rs, x0, offset`	Branch if ≥ zero
`bltz rs, offset`	`blt rs, x0, offset`	Branch if < zero
`bgtz rs, offset`	`blt x0, rs, offset`	Branch if > zero
`bgt rs, rt, offset`	`blt rt, rs, offset`	Branch if >
`ble rs, rt, offset`	`bge rt, rs, offset`	Branch if ≤
`bgtu rs, rt, offset`	`bltu rt, rs, offset`	Branch if >, unsigned
`bleu rs, rt, offset`	`bgeu rt, rs, offset`	Branch if ≤, unsigned
`j offset`	`jal x0, offset`	Jump
`jal offset`	`jal x1, offset`	Jump and link
`jr rs`	`jalr x0, rs, 0`	Jump register
`jalr rs`	`jalr x1, rs, 0`	Jump and link register
`ret`	`jalr x0, x1, 0`	Return from subroutine

`call offset`	`auipc x1, offset[31:12]`
	`jalr x1, x1, offset[11:0]`
		Call far-away subroutine

`tail offset`	`auipc x6, offset[31:12]`
	`jalr x0, x6, offset[11:0]`
		Tail call far-away subroutine

`fence`	`fence iorw, iorw`	Fence on all memory and I/O

Fig 1.4 RISC-V RV32G (**M+I+F+D**) pseudo instructions

1.1 First example: `HelloWorld`

In this first example we are going to write simple HelloWorld program in C language, then we will compile with **`gcc`** it to get the executable code (**`HelloWorld`**) and then assembly code (**`HelloWorld.s`**) .

```
gcc HelloWorld.c -o HelloWorld
```

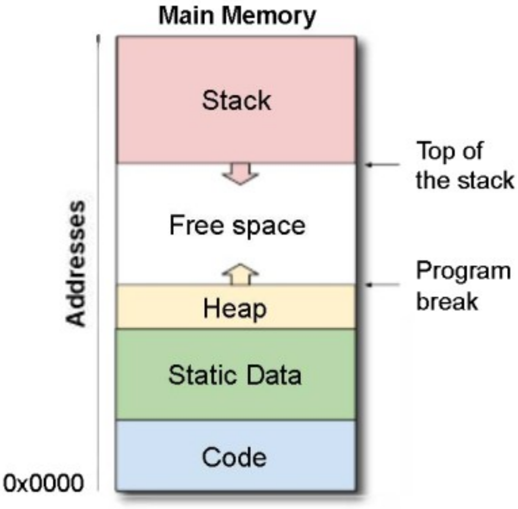
```
// HelloWorld.c
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        printf("HelloWorld\n");
        exit(0);
}
```

Now let us generate assembly code with:

```
gcc HelloWorld.c -c -S -fverbose-asm
```

Now lets us edit the assemble file:

```
        .file   "HelloWorldExit.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "HelloWorld"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
.LFB6:
        .cfi_startproc
        addi    sp,sp,-32
        .cfi_def_cfa_offset 32
        sd      ra,24(sp)
        sd      s0,16(sp)
        .cfi_offset 1, -8
        .cfi_offset 8, -16
        addi    s0,sp,32
        .cfi_def_cfa 8, 0
        mv      a5,a0
        sd      a1,-32(s0)
        sw      a5,-20(s0)
        lla     a0,.LC0
        call    puts@plt
        li      a0,0
        call    exit@plt
        .cfi_endproc
.LFE6:
        .size   main, .-main
        .ident  "GCC: (Debian 13.2.0-4revyos1) 13.2.0"
        .section        .note.GNU-stack,"",@progbits
```



The above code integrates two system calls (call puts@plt and call exit@plt) with the arguments in **`a0: .string`** "**`HelloWorld`**" for **`puts()`** and **`0`** for **`exit()`**.

At the beginning the program decrements the stack pointer (**`-32`**) and stores at the stack the return address (**`sp+24`**), and the stack frame **`s0`** at **`sp+16`**.

It loads to **`a5`** the **`argc`** argument value, and **`argv`** address on the stack.
Then it loads the **`.LC0`** pointer (string) to a0 to pass its value to the **`puts()`** system call.

Note that the program terminates (process) with **exit(0)** not with **return(0)**.

The use of **`return(0)`** would imply the following assembly fragment at the end of the program:

```
        li      a5,0
        mv      a0,a5                   # prepare the retun value - 0
        ld      ra,24(sp)               # restore return address
        ld      s0,16(sp)               # restore s0 pointer
        addi    sp,sp,32                # restore initial stack pointer value
        jr      ra                      # jump to return address
```

The same program can be edited manually, **without the use of stack**:

```
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "HelloWorld"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        mv      a5,a0
        sd      a1,-32(s0)
        sw      a5,-20(s0)
        lla     a0,.LC0
        call    puts@plt             # call puts() system function with .LC0 string
        li      a0,0
        call    exit@plt             # call exit(0) system function
```

1.1.1 First example: `HelloWorldArg` with arguments

```
int main(int argc, char **argv)
{
       printf("argv[1]=%s\n",argv[1]);
        exit(0);
}
```

After the compiling to assembly we obtain (**`HelloWorldArg.s`**):

```
gcc HelloWorldArg.c -c -S
```

```
        .file   "HelloWorldArg.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "argv[1]=%s\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        mv      a5,a0                # move a0 (argc) to a5
        sd      a1,-32(s0)           # a1 (argv address) is stored at s0-32
        sw      a5,-20(s0)           # a5 (a0) is stored at s0-20
        ld      a5,-32(s0)           # a5 receives a1 - argv address
        addi    a5,a5,8              # +8, we are going to read argv[1] not argv[0]
        ld      a5,0(a5)             # load address of argv[1] to a5
        mv      a1,a5                # mv a5 to a1 (for printf)
        lla     a0,.LC0              # load address of string with conversion
        call    printf@plt           # call printf
        li      a0,0                 # load 0 to a0
        call    exit@plt             # call exit(0)
```

The same program without the use of stack:

```
.LC0:
        .string "argv[1]=%s\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
.LFB6:
        mv      a5,a1
        addi    a5,a5,8
        ld      a5,0(a5)
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt
        li      a0,0
        call    exit@plt
```

1.1.2 First example: `HelloWorldArgNum` with **numeric argument**

In this version of **`HelloWorldArgNum`** we are going to capture a **numeric argument** with `argv[1]`.

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        int val=0;
        val=atoi(argv[1]);
        printf("argv[1]=%d\n",val);
        exit(0);
}
```

```
        .file   "HelloWorldArgNum.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "argv[1]=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-48
        sd      ra,40(sp)
        sd      s0,32(sp)
        addi    s0,sp,48
        mv      a5,a0
        sd      a1,-48(s0)
        sw      a5,-36(s0)
        sw      zero,-20(s0)
        ld      a5,-48(s0)
        addi    a5,a5,8
        ld      a5,0(a5)
        mv      a0,a5
        call    atoi@plt
        mv      a5,a0
        sw      a5,-20(s0)
        lw      a5,-20(s0)
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt
        li      a0,0
        call    exit@plt
```

And the same program without the use of stack:

```
.LC0:
        .string "argv[1]=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        mv      a5,a1
        addi    a5,a5,8
        ld      a5,0(a5)
        mv      a0,a5
        call    atoi@plt
        mv      a5,a0
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt
        li      a0,0
        call    exit@plt
```

1.1.3 First example: `HelloWorldArgScan` with `scanf()`

In this version of **`HelloWorldArgScanf`** we are going to read a **numeric argument** with **`scanf()`**.

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int val=0;
        scanf("%d",&val);
        printf("val=%d\n",val);
        exit(0);
}
```

The compiled to assembly with **stack**:

```
        .file   "HelloWorldArgScan.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "%d"
        .align  3
.LC1:
        .string "val=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        sw      zero,-20(s0)
        addi    a5,s0,-20
        mv      a1,a5
        lla     a0,.LC0
        call    __isoc99_scanf@plt
        lw      a5,-20(s0)
        mv      a1,a5
        lla     a0,.LC1
        call    printf@plt
        li      a0,0
        call    exit@plt
```

The same program **without stack (except one address)**:

```
.LC0:
        .string "%d"
        .align  3
.LC1:
        .string "val=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    a5,sp,-8                # here we prepare address for scanf at the stack -8
        mv      a1,a5                   # move address to a1
        lla     a0,.LC0
        call    __isoc99_scanf@plt
        lw      a5,-8(sp)               # load the received converted value at the given address
        mv      a1,a5                   # move it to a1
        lla     a0,.LC1
        call    printf@plt              # call printf with conversion
        li      a0,0
        call    exit@plt
```

1.2 Adding, multiplying and power function

The following examples show how to use some arithmetic operations and branches to multiply or to calculate power function.

1.2.1 `CountToTen` – counting from 9 to 0

Let us take the following example with a simple `while` construct:

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char **argv)
{
        int counter=10;
        while(--counter>=0) printf("Counter=%d\n",counter);
        exit(0);
}
```

```
.LC0:
        .string "Counter=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-48
        sd      ra,40(sp)
        sd      s0,32(sp)
        addi    s0,sp,48
        mv      a5,a0
        sd      a1,-48(s0)
        sw      a5,-36(s0)
        li      a5,10                   # prepare initial value of counter =10
        sw      a5,-20(s0)              # save on stack at s0-20
        j       .L2
.L3:
        lw      a5,-20(s0)              # load saved counter
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt
.L2:
        lw      a5,-20(s0)
        addiw   a5,a5,-1                # decrement counter - a5
        sw      a5,-20(s0)              # save on stack
        lw      a5,-20(s0)
        sext.w  a5,a5                   # sign extend to word
        bge     a5,zero,.L3             # test if greater-equal to zero, then branch if not zero
(false)
        li      a0,0
        call    exit@plt
```

The same program without the stack:

```
.LC0:
        .string "Counter=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        li      a5,10
        j       .L2
.L3:
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt
        mv      a5,s3                   # restore value to a5 from s3
.L2:
        addiw   a5,a5,-1
        mv      s3,a5                   # save new value in s3
        sext.w  a5,a5                   # required for negative value
        bge     a5,zero,.L3             # branch if greater-equal to zero
        li      a0,0                    # load 0 to a0
        addi    a7,x0,93                # load linux function number - exit
        ecall                           # call linux
```

1.2.2 `MultiplyByTen`

The following example with stack is a program that multiplies the fixed integer value (4) by `10`. The multiplication is done uniquely with **addition** and **shift left** instructions.

The C code:

```
#include <stdio.h>
#include <stdlib.h>

int main()
{
        int counter=0,val=4;

        counter=10*val;
        printf("Counter=%d\n",counter);
        exit(0);
}
```

The compiled assembly code:

```
.LC0:
        .string "Counter=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        sw      zero,-20(s0)
        li      a5,4
        sw      a5,-24(s0)
        lw      a5,-24(s0)
        mv      a4,a5
        mv      a5,a4
        slliw   a5,a5,2
        addw    a5,a5,a4
        slliw   a5,a5,1
        sw      a5,-20(s0)
        lw      a5,-20(s0)
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt
        li      a0,0
        call    exit@plt
```

Note that the multiplication is done with **shift left** and **add** instructions only:

```
        slliw   a5,a5,2                       # shift left 2 bits - multiply a5 (4) by 4: 16
        addw    a5,a5,a4                      # add a4 (4) to a5: 16+4 :20
        slliw   a5,a5,1                       # shift left 1 bit - multiply by 2: 40
```

The same program without the use of stack:

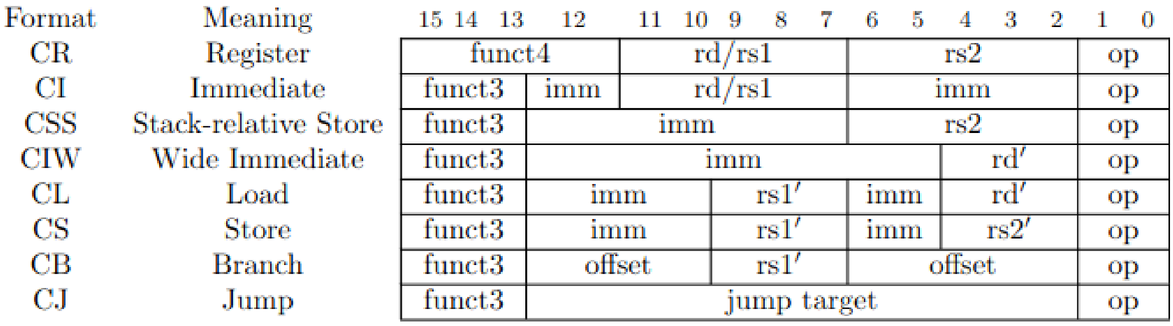
```
.LC0:
        .string "Counter=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        li      a5,4
        mv      a4,a5
        mv      a5,a4
        slliw   a5,a5,2
        addw    a5,a5,a4
        slliw   a5,a5,1
        mv      a1,a5
        lla     a0,.LC0
        call    printf@plt
        li      a0,0
        call    exit@plt
```

1.2.2.1 Binary code analyzis

Now let us try to analyze the part of the generated **binary code** belonging to `main` **section**.
We can **disassembly** code into **binary dump**:

```
ubuntu@ubuntu:~/assembly$ riscv64-linux-gnu-objdump -d MultiplyByTenNoStack
```

Let us extract just the `main` section that contains **32-bit** and **16-bit compressed format** instructions.



<table>
<tr><th>Format</th><th>Meaning</th><th>15</th><th>14</th><th>13</th><th>12</th><th>11</th><th>10</th><th>9</th><th>8</th><th>7</th><th>6</th><th>5</th><th>4</th><th>3</th><th>2</th><th>1</th><th>0</th></tr>
<tr><td>CR</td><td>Register</td><td colspan="4">funct4</td><td colspan="5">rd/rs1</td><td colspan="5">rs2</td><td colspan="2">op</td></tr>
<tr><td>CI</td><td>Immediate</td><td colspan="3">funct3</td><td>imm</td><td colspan="5">rd/rs1</td><td colspan="5">imm</td><td colspan="2">op</td></tr>
<tr><td>CSS</td><td>Stack-relative Store</td><td colspan="3">funct3</td><td colspan="6">imm</td><td colspan="5">rs2</td><td colspan="2">op</td></tr>
<tr><td>CIW</td><td>Wide Immediate</td><td colspan="3">funct3</td><td colspan="8">imm</td><td colspan="3">rd′</td><td colspan="2">op</td></tr>
<tr><td>CL</td><td>Load</td><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rs1′</td><td colspan="2">imm</td><td colspan="3">rd′</td><td colspan="2">op</td></tr>
<tr><td>CS</td><td>Store</td><td colspan="3">funct3</td><td colspan="3">imm</td><td colspan="3">rs1′</td><td colspan="2">imm</td><td colspan="3">rs2′</td><td colspan="2">op</td></tr>
<tr><td>CB</td><td>Branch</td><td colspan="3">funct3</td><td colspan="3">offset</td><td colspan="3">rs1′</td><td colspan="5">offset</td><td colspan="2">op</td></tr>
<tr><td>CJ</td><td>Jump</td><td colspan="3">funct3</td><td colspan="11">jump target</td><td colspan="2">op</td></tr>
</table>

```
00000000000006c4 <main>:
 6c4:   4791                    li      a5,4                 # compressed load immediate
 6c6:   873e                    mv      a4,a5                # compressed move
 6c8:   87ba                    mv      a5,a4                # compressed move
 6ca:   0027979b                sllw    a5,a5,0x2
 6ce:   9fb9                    addw    a5,a5,a4             # compressed add register
 6d0:   0017979b                sllw    a5,a5,0x1
 6d4:   85be                    mv      a1,a5                # compressed move
 6d6:   00000517                auipc   a0,0x0
 6da:   fe250513                addi    a0,a0,-30
 6de:   f13ff0ef                jal     5f0 <printf@plt>
 6e2:   4501                    li      a0,0                 # compressed load immediate
 6e4:   efdff0ef                jal     5e0 <exit@plt>
```

Let us try to understand some of the above binary codes.

```
873e                    mv    a4,a5             # compressed move
1000 01110 01111 10: funct3=1000 (mv), 01110 (14-a4), 01111 (15-a5), op=10 (R)
```

```
addi -  01e50513

000000011110 01010 000 01010 0010011
   imm        rd        rs1   code
    30      a0-x10    a0-x10  addi

sllw  a5,a5,0x1  : sllw -  0017979b

000000000001 01111 001 01111 0011011
    imm       rd   fun   rs1   code
      1     a5-x15 SL  a5-x15  sllw
```

1.2.3 `MultiplyByHundred`

In this example we use **directly** the **multiplication** instruction:

```
mulw    a5,a4,a5
```

This is the complete code written directly **without stack**. To capture the input data (multiplicand) we use **`scanf()`**:

```
        .file   "MultHundredScan.s"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "%d"
        .align  3
.LC1:
        .string "a=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    a5,sp,-8                # here we prepare address for scanf at the stack -4
        mv      a1,a5                   # argument-address for scanf
        lla     a0,.LC0                 # format string for scanf
        call    scanf@plt
        lw      a5,-8(sp)               # load the received  value from scanf
        li      a4,100
        mulw    a5,a5,a4
        mv      a1,a5                   # here is the value to be printed
        lla     a0,.LC1                 # format for printf
        call    printf@plt
        li      a0,0                    # load return value
        addi    a7,x0,93                # linux exit call number
        call    exit@plt
```

1.2.4 Calculating with `fpow()` function

In this example we study `while` **loop** construct for **power calculation function**. This is C code for `fpow()` function.

```
#include <stdio.h>
#include <stdlib.h>

int fpow(int x, int y)
{
        int res=1,i=0;
        while(i<y)
                {
                res=x*res; i++;
                }
        return res;
}

int main(int argc, char **argv)
{
        int val,base,exp;
        base=atoi(argv[1]);
        exp=atoi(argv[2]);
        val=fpow(base,exp);
        printf("val=%d\n",val);
}
```

This C code may be compiled to assembly code:

```
gcc PowerFun.c -c -S
```

To get the corresponding assembly code:

```
        .file   "PowerFun.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  fpow
        .type   fpow, @function
fpow:
        addi    sp,sp,-48
        sd      s0,40(sp)
        addi    s0,sp,48
        mv      a5,a0
        mv      a4,a1
        sw      a5,-36(s0)              # save first argument base at s0-36
        mv      a5,a4
        sw      a5,-40(s0)              # save second argument: exponent at s0-40
        li      a5,1                    # set initial result =1
        sw      a5,-20(s0)
        sw      zero,-24(s0)
        j       .L2                     # jump to .L2
.L3:
        lw      a5,-20(s0)
        mv      a4,a5
        lw      a5,-36(s0)
        mulw    a5,a4,a5                # multiply partial result by base at s0-36
        sw      a5,-20(s0)
        lw      a5,-24(s0)
        addiw   a5,a5,1                 # increment partial exponent by 1
        sw      a5,-24(s0)              # save partial exponent at s0-24
.L2:
        lw      a5,-24(s0)
        mv      a4,a5                   # get partial exponent to a4
        lw      a5,-40(s0)              # get final exponent value
        sext.w  a4,a4                   # sign extend if negative !
        sext.w  a5,a5
        blt     a4,a5,.L3               # branch to .L3 if partial smaller than final
        lw      a5,-20(s0)              # load the final result
        mv      a0,a5                   # move final result to a0
        ld      s0,40(sp)
        addi    sp,sp,48
        jr      ra                      # return to main
.LC0:
        .string "val=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-48
        sd      ra,40(sp)
        sd      s0,32(sp)
        addi    s0,sp,48
        mv      a5,a0
        sd      a1,-48(s0)              # save a1 - argv[] at s0-48
        sw      a5,-36(s0)
        ld      a5,-48(s0)              # load argv[] to a5
        addi    a5,a5,8                 # add 8 to get argv[1] address
        ld      a5,0(a5)
        mv      a0,a5
        call    atoi@plt                # call atoi() to convert argv[1]
        mv      a5,a0
        sw      a5,-20(s0)
        ld      a5,-48(s0)
        addi    a5,a5,16                # add 16 to get argv[2] address
        ld      a5,0(a5)
        mv      a0,a5
        call    atoi@plt                # call atoi() to convert argv[2]
        mv      a5,a0
        sw      a5,-24(s0)
        lw      a4,-24(s0)
        lw      a5,-20(s0)
        mv      a1,a4
        mv      a0,a5
        call    fpow                    # call fpow with two arguments in a1, a0
        mv      a5,a0                   # move result to a5
        sw      a5,-28(s0)
        lw      a5,-28(s0)
        mv      a1,a5                   # pass result to printf
        lla     a0,.LC0
        call    printf@plt
        li      a5,0
        mv      a0,a5
        ld      ra,40(sp)
        ld      s0,32(sp)
        addi    sp,sp,48
        jr      ra
```

1.2.4.1 Simplified `power` function without stack and and fixed arguments

```
        .file   "pow.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  power
        .type   power, @function
power:
        mv      a4,a1                # first argument - base
        mv      a5,a2                # second argument - exponent
        li      a6,1                 # initial result base^0
        j       .L2                  # jump to starting block - L4
.L4:
        mv      a7,a5                # move a5 to a7 for mask test
        andi    a7,a7,1              # mask test 0 or 1
        beq     a7,zero,.L3          # if zero go to L3
        mulw    a6,a6,a4             # else multiply the base with partial result
.L3:
        mulw    a4,a4,a4             # multiply the base
        srli    a5,a5,1              # shift right the exponent
.L2:
        bne     a5,zero,.L4          # test the value of shift exponent, if non zero go to L4
        jr      ra                   # else return to main block
        .size   power, .-power
        .section        .rodata
        .align  3
.LC0:
        .string "2^8=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        li      a1,2                    # prepare base value
        li      a2,8                    # prepare exponent value
        call    power                   # call power block
        mv      a1,a6                   # move result to a1
        lla     a0,.LC0                 # load string address to a0
        call    printf@plt              # call linux printf
        li      a0,0                    # load 0 to a0
        addi    a7,x0,93                # load linux function number - exit
        ecall                           # call linux
```

To do:

1. Analyze the above assembly code and **compare it with the corresponding C program**. Why **we do not need** the addition or subtraction instructions !

```
#include <stdio.h>
/* Iterative Function to calculate (x^y) in O(logy) */
int power(int x, unsigned int y)
{
    int res = 1; // Initialize result
    while (y > 0) {
        // If y is odd, multiply the partial result with x
        if (y & 1)  // test the last bit of y
            res = res * x;
        // n must be even now
        y = y >> 1; // y = y/2
        x = x * x; //
    }
    return res;
}

int main()
{
        int pow=0;
        pow=power(2,8);
        printf("2^8=%d\n",pow);
}
```

Try to understand the above code where the exponent value binary coded is shifted **one bit right at every step.**
Take another example: `x=101` and `y=101` (`5^5`)

1.2.4.2 Simplified `power` function without stack and program arguments

```
       .file   "PowerScanArgs.s"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  power
        .type   power, @function
power:
        mv      a4,a1                      # first argument - base
        mv      a5,a2                      # second argument - exponent
        li      a6,1                       # initial result base^0
        j       .L2                        # jump to starting block - L4
.L4:
        mv      a7,a5                      # move a5 to a7 for mask test
        andi    a7,a7,1         # mask test 0 or 1
        beq     a7,zero,.L3                # if zero go to L3
        mulw    a6,a6,a4                   # else multiply the base with partial result
.L3:
        mulw    a4,a4,a4                   # multiply the base
        srli    a5,a5,1         # shift right the exponent
```

```
.L2:
        bne     a5,zero,.L4             # test the value of shift exponent, if non zero go to L4
        jr      ra                      # else return to main block
        .size   power, .-power
        .section        .rodata
        .align  3
.LC0:
        .string "res=%d\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
#       mv      s0,a0                   # save argument argc
        mv      s1,a1                   # save argument argv[] in s1

        mv      a5,s1
        addi    a5,a5,8                 # first argument in argv[1]
        ld      a5,0(a5)
        mv      a0,a5
        call    atoi@plt
        mv      s2,a0                   # saved first argument value (base)

        mv      a5,s1
        addi    a5,a5,16                # second argument in argv[2]
        ld      a5,0(a5)
        mv      a0,a5
        call    atoi@plt
        mv      s3,a0                   # saved second argument value (exp)

        mv      a1,s2                   # base
        mv      a2,s3                   # exp
        call    power                   # call power block
        mv      a1,a6                   # move result to a1
        lla     a0,.LC0                 # load string address to a0
        call    printf@plt              # call linux printf
        li      a0,0                    # load 0 to a0
        addi    a7,x0,93                # load linux function number - exit
        ecall                           # call linux
```

To do

1. Analyze the program and explain the **implementation** of program arguments.
2. Detect the absence of arguments, and it is the case do **`exit(1)`** from the program.

Lab 2

System and Network functions

In this lab we are going to program some examples using system call related to interruptions, files and network protocols.

2.1 Processing interruptions (signals)

The following example shows how to capture **^C** signal (`SIGINT`) sent from the keyboard.

```
#include  <signal.h>
#include  <stdio.h>
#include  <unistd.h>

void mysig(int sig)
{
printf("SIGINT\n");
}

int main()
{
        signal(SIGINT,mysig);
        while(1)
        {
                sleep(3);
        }
}
```

Compile the above C code with:

```
gcc SignalSigint.c –o SignalSigint
```

and run it:

```
bako@lpi4a:~/Programming/lab2$ ./SignalSigint
^CSIGINT
^CSIGINT
^CSIGINT
^Z
[1]+  Stopped                 ./SignalSigint
```

Note that **you have to stop it** with `^Z` signal (`^C` is captured).

```
bako@lpi4a:~/Programming/lab2$ ps
    PID TTY          TIME CMD
   1091 pts/2    00:00:00 bash
   1357 pts/2    00:00:00 SignalSigint
   1358 pts/2    00:00:00 ps
```

Now you can kill it with `SIGKILL` (**9**):

```
bako@lpi4a:~/Programming/lab2$ kill -9 1357
```

And the corresponding assembly code.

```
        .file   "int_sig2.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "SIGINT"
        .text
        .align  1
        .globl  mysig
        .type   mysig, @function
mysig:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        mv      a5,a0
        sw      a5,-20(s0)
        lla     a0,.LC0
        call    puts@plt             # replace it with printf(“%d\n”,n), n is mysig argument
        nop
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
        .size   mysig, .-mysig
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-16
        sd      ra,8(sp)
        sd      s0,0(sp)
        addi    s0,sp,16
        lla     a1,mysig
        li      a0,2
        call    signal@plt
.L3:
        li      a0,3
        call    sleep@plt
        j       .L3
        .size   main, .-main
        .ident  "GCC: (Ubuntu 12.2.0-3ubuntu1) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

To do

1. Analyze the above codes
2. Write the same program without the use of stack
3. In assembler code in `mysig:` function replace `puts` by `printf` call to print the **`SIGINT` signal number** (received in `a0`).

2.2 Reading from and writing (creating) to files

2.2.1 Read text file and printing to `stdout` with buffer space on stack

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
       int fd,nb; char buff[512];
       fd=open(argv[1],0);
       nb=read(fd,buff,512);
       write(1,buff,nb);
       close(fd);
       return 0;
}
```

```
bako@licheerv:~/Programs/ass$ gcc ReadFile.c -o ReadFile
bako@licheerv:~/Programs/ass$ ./ReadFile ReadFile.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
       int fd,nb; char buff[512];
       fd=open(argv[1],0);
       nb=read(fd,buff,512);
       write(1,buff,nb);
       close(fd);
       return 0;
}
```

After the compilation with:

```
       gcc ReadFile.c -o -S
```

We obtain the following assembly code:

```
        .file   "ReadFile.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-560                # preparing new stack pointer: sp=sp-560
        sd      ra,552(sp)                # saving return address on stack
        sd      s0,544(sp)                # saving frame s0 on stack
        addi    s0,sp,560                 # preparing new stack frame in s0
        mv      a5,a0                     # moving a0 to a5
        sd      a1,-560(s0)               # saving a1 (second argument) at s0-560
        sw      a5,-548(s0)               # saving a0,a5 (first argument) at s0-548
        ld      a5,-560(s0)               # loading a0 from s0-560 (a0)
        addi    a5,a5,8                   # adding 8 to a5
        ld      a5,0(a5)                  # loading a5 from a5+0
        li      a1,0                      # loading 0 to a1
        mv      a0,a5                     # moving a5 to a0
        call    open@plt                  # call with name in a0 and mode in a1 (0)
        mv      a5,a0                     # move a0 (file pointer= 3?) to a5
        sw      a5,-20(s0)                # save a5 at s0-20
        addi    a4,s0,-536                # add to a4+s0-536 (buffer address on stack)
        lw      a5,-20(s0)                # load s0-20 to a5 (file pointer)
        li      a2,512                    # required size of file segment (512)
        mv      a1,a4                     # load buffer address in a4 to a1
        mv      a0,a5                     # load file pointer in a5 to a0
        call    read@plt                     # call read system function
        mv      a5,a0                        # move a0 (number of received bytes) to a5
        sw      a5,-24(s0)                   # save a5 at s0-24
        lw      a4,-24(s0)                   # load to a4 from s0-24 (number of bytes)
        addi    a5,s0,-536                   # add to a5 s0-536 : new pointer to read file segment
        mv      a2,a4                        # move a4 to a2 (number of bytes to write)
        mv    with buffer space on stack  a1,a5                              # load a5 (text address in
buffer) to a1
        li      a0,1                         # load file pointer : 1 stdio to a0
        call    write@plt                    # write to stdio (terminal window)
        lw      a5,-20(s0)                   # load a5 from s0-20 (file pointer)
        mv      a0,a5                        # move a5 to a0
        call    close@plt                    # close file
        li      a5,0                         # load 0 to a5
        mv      a0,a5                        # move a5 to a0
        ld      ra,552(sp)                   # load return address from sp+552
        ld      s0,544(sp)                   # load previous frame s0 from sp+544
        addi    sp,sp,560                    # reload stack pointer address
        jr      ra                           # return from main
        .size   main, .-main
        .ident  "GCC: (Debian 12.2.0-10) 12.2.0"
```

Note that the program use of **stack** allows to allocate the space for the buffer (`char buff[512];`). The stack provides **560 bytes including the buffer and other variables**.

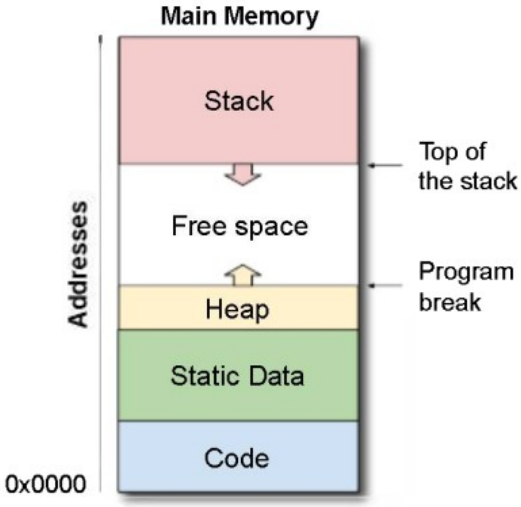
2.2.2 Read text file and printing to `stdout` with buffer space in `global` region

Now let us allocate the **data space globally** and see the behavior of the compiled program.

```
bako@licheerv:~/Programs/ass$ gcc ReadFileGlob.c -o ReadFileGlob

bako@licheerv:~/Programs/ass$ ./ReadFileGlob ReadFileGlob.c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

char buff[512];
int fd,nb;

int main(int argc, char **argv)
{
       fd=open(argv[1],0);
       nb=read(fd,buff,512);
       write(1,buff,nb);
       close(fd);
       return 0;
}
```

Let us see the generated assembly code:

```
       .file   "ReadFileGlob.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .globl  buff
        .bss
        .align  3
        .type   buff, @object
        .size   buff, 512
buff:
        .zero   512
        .globl  fd
        .align  2
        .type   fd, @object
        .size   fd, 4
fd:
        .zero   4
        .globl  nb
        .align  2
        .type   nb, @object
        .size   nb, 4
nb:
        .zero   4
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-32
        sd      ra,24(sp)
        sd      s0,16(sp)
        addi    s0,sp,32
        mv      a5,a0
        sd      a1,-32(s0)
        sw      a5,-20(s0)
        ld      a5,-32(s0)
        addi    a5,a5,8
        ld      a5,0(a5)
        li      a1,0
        mv      a0,a5
        call    open@plt
        mv      a5,a0
        mv      a4,a5
        lla     a5,fd
        sw      a4,0(a5)
        lla     a5,fd
        lw      a5,0(a5)
        li      a2,512
        lla     a1,buff
        mv      a0,a5
        call    read@plt
        mv      a5,a0
        sext.w  a4,a5
        lla     a5,nb
        sw      a4,0(a5)
        lla     a5,nb
        lw      a5,0(a5)
        mv      a2,a5
        lla     a1,buff
        li      a0,1
        call    write@plt
        lla     a5,fd
        lw      a5,0(a5)
        mv      a0,a5
        call    close@plt
        li      a5,0
        mv      a0,a5
        ld      ra,24(sp)
        ld      s0,16(sp)
        addi    sp,sp,32
        jr      ra
        .size   main, .-main
        .ident  "GCC: (Debian 12.2.0-14) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```



In this case the stack occupies only 32 bytes. The declared variables are placed in **global** segment (**`.glob`**):

```
        .globl  buff
        .bss
        .align  3
        .type   buff, @object
        .size   buff, 512
```

To do:

1. Analyze the above codes
2. In assembler code remove the use of stack and modify the size of the buffer from **512 to 2048** and the corresponding **`read`** call.

```
        .file   "ReadFileGlobNoStack.s"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .text
        .globl  buff
        .bss
        .align  3
        .type   buff, @object
        .size   buff, 2048
buff:
        .zero   2048
        .globl  fd
        .align  2
        .type   fd, @object
        .size   fd, 4
fd:
        .zero   4
        .globl  nb
        .align  2
        .type   nb, @object
        .size   nb, 4
nb:
        .zero   4
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        mv      a5,a1
        addi    a5,a5,8
        ld      a5,0(a5)
        li      a1,0
        mv      a0,a5
        call    open@plt
        mv      a5,a0
        mv      a4,a5
        lla     a5,fd
        sw      a4,0(a5)
        lla     a5,fd
        lw      a5,0(a5)
        li      a2,2048
        lla     a1,buff
        mv      a0,a5
        call    read@plt
        mv      a5,a0
        sext.w  a4,a5
        lla     a5,nb
        sw      a4,0(a5)
        lla     a5,nb
        lw      a5,0(a5)
        mv      a2,a5
        lla     a1,buff
        li      a0,1
        call    write@plt
        lla     a5,fd
        lw      a5,0(a5)
        mv      a0,a5
        call    close@plt
        li      a0,0                 # load 0 to a0
        addi    a7,x0,93             # load linux function number - exit
        ecall                        # call linux to retun from program
        .size   main, .-main
        .ident  "GCC: (Debian 12.2.0-14) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

Compile the assembly program and run it as follows:

```
bako@lpi4a:~/Programming/lab2$ gcc ReadFileGlobNoStack.s -o ReadFileGlobNoStack
bako@lpi4a:~/Programming/lab2$ ./ReadFileGlobNoStack ReadFileGlobNoStack.s
```

2.2.3 Read text file and printing to `stdout` with buffer space in `heap` region

The following example is the same as the previous one with one exception, the buffer space is not global, it is allocated dynamically by **`malloc()`** function.

```
#include <stdio.h> #include <stdlib.h>
#include <sys/types.h> #include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
        int fd,nb; char *buff;
        buff=malloc(512);fd=open(argv[1],0);
        nb=read(fd,buff,512); write(1,buff,nb);
        close(fd);
        return 0;
}
```

This is corresponding assembly program:

```
        .file   "ReadFileMalloc.s"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-48
        sd      ra,40(sp)
        sd      s0,32(sp)
        addi    s0,sp,48
        mv      a5,a0
        sd      a1,-48(s0)
        sw      a5,-36(s0)
        li      a0,512
        call    malloc@plt              # call malloc(512)
        mv      a5,a0                   # buff address in a0
        sd      a5,-24(s0)              # buff address stored at s0-24
        ld      a5,-48(s0)
        addi    a5,a5,8
        ld      a5,0(a5)
        li      a1,0
        mv      a0,a5
        call    open@plt
        mv      a5,a0                   # file pointer in a0 to a5
        sw      a5,-28(s0)              # file pointer stored at s0-28
        lw      a5,-28(s0)
        li      a2,512                  # buff size in a2
        ld      a1,-24(s0)              # buff address from s0-24 in a1
        mv      a0,a5                   # file pointer in a0
        call    read@plt
        mv      a5,a0                   # number of bytes or error
        sw      a5,-32(s0)
        lw      a5,-32(s0)
        mv      a2,a5                   # number of bytes in a2
        ld      a1,-24(s0)              # buffer address at s0-24
        li      a0,1                    # file pointer (stdout) in a0
        call    write@plt
        lw      a5,-28(s0)
        mv      a0,a5
        call    close@plt
        li      a5,0
        mv      a0,a5
        ld      ra,40(sp)
        ld      s0,32(sp)
        addi    sp,sp,48
```

To do:

Modify the size of the buffer allocated by **`malloc()`** to **2048** bytes.

2.2.4 Copy text or binary file

The following C code copies the source file to destination file. The copy is done in a loop with `while` and **512-byte buffer**.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char **argv)
{
        int fdr,fdw,nb; char buff[512];
        fdr=open(argv[1],0);
        fdw=creat(argv[2],0777);
        while(nb=read(fdr,buff,512)) write(fdw,buff,nb);
        close(fdr);close(fdw);
        return 0;
}
```

After the compilation with:

```
        gcc CopyFile -c -S
```

we obtain the following assembly code:

```
        .file   "CopyFile.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-560
        sd      ra,552(sp)
        sd      s0,544(sp)
        addi    s0,sp,560
        mv      a5,a0
        sd      a1,-560(s0)
        sw      a5,-548(s0)
        ld      a5,-560(s0)
        addi    a5,a5,8
        ld      a5,0(a5)
        li      a1,0
        mv      a0,a5
        call    open@plt
        mv      a5,a0
        sw      a5,-20(s0)
        ld      a5,-560(s0)
        addi    a5,a5,16
        ld      a5,0(a5)
        li      a1,511
        mv      a0,a5
        call    creat@plt
        mv      a5,a0
        sw      a5,-24(s0)
        j       .L2
.L3:
        lw      a3,-28(s0)
        addi    a4,s0,-544
        lw      a5,-24(s0)
        mv      a2,a3
        mv      a1,a4
        mv      a0,a5
        call    write@plt
.L2:
        addi    a4,s0,-544
        lw      a5,-20(s0)
        li      a2,512
        mv      a1,a4
        mv      a0,a5
        call    read@plt
        mv      a5,a0
        sw      a5,-28(s0)
        lw      a5,-28(s0)
        sext.w  a5,a5
        bne     a5,zero,.L3
        lw      a5,-20(s0)
        mv      a0,a5
        call    close@plt
        lw      a5,-24(s0)
        mv      a0,a5
        call    close@plt
        li      a5,0
        mv      a0,a5
        ld      ra,552(sp)
        ld      s0,544(sp)
        addi    sp,sp,560
        jr      ra
        .size   main, .-main
        .ident  "GCC: (Debian 12.2.0-10) 12.2.0"
        .section        .note.GNU-stack,"",@progbits
```

Link edition and execution example:

```
gcc CopyFile.s -o CopyFile

./CopyFile CopyFile.c CopyFile_bat.c
```

To do

1. Analyze the above codes
2. Use the global space for buffer and variables
3. Remove the use of stack

2.3 Network functions and protocols

In this part of laboratory we are presenting a few simple examples to send and receive text messages with UDP and TCP protocol.

2.3.1 Sending and receiving UDP messages

2.3.1.1 Sending UDP datagrames

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>
#define BUFLEN 512      // Max length of buffer
#define PORT 8888       // The port on which to send or listen for data
#define REMOTE_ADDR  "192.168.1.72"

int main(void)
{
  struct sockaddr_in si_me, si_other;
  int s;
  char buf[BUFLEN];
  s=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
  // this structure and bind() are optional for first sender
  // the receiver can extract the sender socket
  memset((char *) &si_me,0x00,16);
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  // bind socket to port
  bind(s,(struct sockaddr*)&si_me,16);
  // zero out the structure other (receiver) - this structure is mandatory
  memset((char *)&si_other,0x00,16);
  si_other.sin_family = AF_INET;
  si_other.sin_port = htons(PORT);  // to work globally
//si_other.sin_port = htons(PORT+1);  // to work locally
//si_other.sin_addr.s_addr = htonl(INADDR_ANY); // to work locally
  si_other.sin_addr.s_addr = inet_addr(REMOTE_ADDR); // to work globally
  //keep sending data
  while(1)
    {
    printf("write the message\n");scanf("%s",buf);
    // try to send data, this is a non blocking call
    sendto(s,buf,strlen(buf)+1,0,(struct sockaddr *)&si_other,16);
    }
  close(s);
  return 0;
}
```

After the compiling with:

```
gcc UdpSend.s -c -S
```

We obtain the following assembly code:

```
        .file   "Send.UDP.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "192.168.1.72"
        .align  3
.LC1:
        .string "write the message"
        .align  3
.LC2:
        .string "%s"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-576
        sd      ra,568(sp)
        sd      s0,560(sp)
        addi    s0,sp,576               # 576 bytes on stack
        li      a2,17                   # UDP protocol number
        li      a1,2
        li      a0,2
        call    socket@plt
        mv      a5,a0
        sw      a5,-20(s0)
        addi    a5,s0,-40
        li      a2,16
        li      a1,0
        mv      a0,a5
        call    memset@plt
        li      a5,2
        sh      a5,-40(s0)
        li      a5,8192                 # load 0x2000
        addi    a0,a5,696               # add 696 to get 8888
        call    htons@plt
        mv      a5,a0
        sh      a5,-38(s0)
        li      a0,0
        call    htonl@plt
        mv      a5,a0
        sext.w  a5,a5
        sw      a5,-36(s0)
        addi    a4,s0,-40
        lw      a5,-20(s0)
        li      a2,16
        mv      a1,a4
        mv      a0,a5
        call    bind@plt
        addi    a5,s0,-56
        li      a2,16
        li      a1,0
        mv      a0,a5
        call    memset@plt
        li      a5,2
        sh      a5,-56(s0)
        li      a5,8192
        addi    a0,a5,696
        call    htons@plt
        mv      a5,a0
        sh      a5,-54(s0)
        lla     a0,.LC0                 # internet address in a0
        call    inet_addr@plt
        mv      a5,a0
        sext.w  a5,a5
        sw      a5,-52(s0)
.L2:
        lla     a0,.LC1
        call    puts@plt
        addi    a5,s0,-568
        mv      a1,a5
        lla     a0,.LC2
        call    __isoc99_scanf@plt
        addi    a5,s0,-568
        mv      a0,a5
        call    strlen@plt
        mv      a5,a0
        addi    a2,a5,1
        addi    a4,s0,-56
        addi    a1,s0,-568
        lw      a0,-20(s0)
        li      a5,16
        li      a3,0
        call    sendto@plt
        j       .L2
```

To do:

1. Analyze the above code.
2. Modify the IP address and port number.
3. Modify the stack use by implementing the **`buffer`** in **global** region.

2.3.1.2 Receiving UDP datagrames

The following C code allows us to receive the UDP packets from the above explained sender.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <arpa/inet.h>
#include <sys/socket.h>

#define BUFLEN 512      //Max length of buffer
#define PORT 8889       //The port on which to listen for incoming data

int main(void)
{
  struct sockaddr_in si_me, si_other;
  int s, i, slen = 16,rlen;
  char buf[BUFLEN];
  //create a UDP socket
  s=socket(AF_INET,SOCK_DGRAM,IPPROTO_UDP);
  // zero out the structure
  memset((char *) &si_me, 0, 16);
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  //bind socket to port
  bind(s,(struct sockaddr*)&si_me,16);
  //keep listening for data
  while(1)
    {
    printf("Waiting for data...");
    //try to receive some data, this is a blocking call
    rlen=recvfrom(s,buf,BUFLEN,0,(struct sockaddr *)&si_other,&slen);
    //print details of the client/peer and the data received
    printf("IP, port %s:%d\n",inet_ntoa(si_other.sin_addr), ntohs(si_other.sin_port));
    printf("Received data: %s\n", buf);
    }
  close(s);
  return 0;
}
```

After the compiling to assembly we obtain:

```
        .file   "Recv.UDP.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "Waiting for data..."
        .align  3
.LC1:
        .string "IP, port %s:%d\n"
        .align  3
.LC2:
        .string "Received data: %s\n"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-592
        sd      ra,584(sp)
        sd      s0,576(sp)
        sd      s1,568(sp)
        addi    s0,sp,592
        li      a5,16
        sw      a5,-76(s0)
        li      a2,17
        li      a1,2
        li      a0,2
        call    socket@plt
        mv      a5,a0
        sw      a5,-36(s0)
        addi    a5,s0,-56
        li      a2,16
        li      a1,0
        mv      a0,a5
        call    memset@plt
        li      a5,2
        sh      a5,-56(s0)
        li      a5,8192
        addi    a0,a5,697
        call    htons@plt
        mv      a5,a0
        sh      a5,-54(s0)
        li      a0,0
        call    htonl@plt
        mv      a5,a0
        sext.w  a5,a5
        sw      a5,-52(s0)
        addi    a4,s0,-56
        lw      a5,-36(s0)
        li      a2,16
        mv      a1,a4
        mv      a0,a5
        call    bind@plt
.L2:
        lla     a0,.LC0
        call    printf@plt
        addi    a5,s0,-76
        addi    a4,s0,-72
        addi    a1,s0,-592
        lw      a0,-36(s0)
        li      a3,0
        li      a2,512
        call    recvfrom@plt
        mv      a5,a0
        sw      a5,-40(s0)
        lw      a5,-68(s0)
        mv      a0,a5
        call    inet_ntoa@plt
        mv      s1,a0
        lhu     a5,-70(s0)
        mv      a0,a5
        call    ntohs@plt
        mv      a5,a0
        sext.w  a5,a5
        mv      a2,a5
        mv      a1,s1
        lla     a0,.LC1
        call    printf@plt
        addi    a5,s0,-592
        mv      a1,a5
        lla     a0,.LC2
        call    printf@plt
        j       .L2
```

To do:

1. Analyze the above code.
2. Modify the IP address and port number.
3. Execute together with the sender code on the same and on separate hosts
4. Modify the stack use by implementing the **buffer** in **global** region.

2.3.2 Sending and receiving TCP segments

2.3.2.1 The sender code with `connect()`

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFLEN 512      // Max length of buffer
#define PORT 8898       // the port on which to send or listen for data
#define REMOTE_ADDR  "192.168.1.72"

int main(void)
{
  struct sockaddr_in si_me, si_other;
  int s, i, slen = sizeof(si_other) , recv_len, res;
  char buf[BUFLEN];
  s = socket(AF_INET, SOCK_STREAM, 0);
  memset((char *) &si_other, 0, sizeof(si_other));
  si_other.sin_family = AF_INET;
  //si_other.sin_port = htons(PORT+1);  // to work locally
  si_other.sin_port = htons(PORT);  // to work globally
  //si_other.sin_addr.s_addr = htonl(INADDR_ANY); // to work locally
  si_other.sin_addr.s_addr = inet_addr(REMOTE_ADDR); // to work globally
  res=connect(s,(struct sockaddr *)&si_other,sizeof(si_other));
  if(res<0) { printf("No connection\n");exit(1); }
    while(1)
      {
      memset(buf,0x00,BUFLEN);
      printf("write the message, replace spaces by _, end message with . \n");scanf("%s",buf);
      write(s,buf,strlen(buf)+1);
      sleep(3);
      if(buf[0]=='.') break;
      }
  close(s);
}
```

And the corresponding assembly code:

```
        .file   "Send.TCP.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "192.168.1.72"
        .align  3
.LC1:
        .string "No connection"
        .align  3
.LC2:
        .string "write the message, replace spaces by _, end message with . "
        .align  3
.LC3:
        .string "%s"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-576
        sd      ra,568(sp)
        sd      s0,560(sp)
        addi    s0,sp,576
        li      a5,16
        sw      a5,-20(s0)
        li      a2,0
        li      a1,1
        li      a0,2
        call    socket@plt
        mv      a5,a0
        sw      a5,-24(s0)
        addi    a5,s0,-64
        li      a2,16
        li      a1,0
        mv      a0,a5
        call    memset@plt
        li      a5,2
        sh      a5,-64(s0)
        li      a5,8192
        addi    a0,a5,706
        call    htons@plt
        mv      a5,a0
        sh      a5,-62(s0)
        lla     a0,.LC0
        call    inet_addr@plt
        mv      a5,a0
        sext.w  a5,a5
        sw      a5,-60(s0)
        addi    a4,s0,-64
        lw      a5,-24(s0)
        li      a2,16
        mv      a1,a4
        mv      a0,a5
        call    connect@plt
        mv      a5,a0
        sw      a5,-28(s0)
        lw      a5,-28(s0)
        sext.w  a5,a5
        bge     a5,zero,.L2
        lla     a0,.LC1
        call    puts@plt
        li      a0,1
        call    exit@plt
.L2:
        addi    a5,s0,-576
        li      a2,512
        li      a1,0
        mv      a0,a5
        call    memset@plt
        lla     a0,.LC2
        call    puts@plt
        addi    a5,s0,-576
        mv      a1,a5
        lla     a0,.LC3
        call    __isoc99_scanf@plt
        addi    a5,s0,-576
        mv      a0,a5
        call    strlen@plt
        mv      a5,a0
        addi    a3,a5,1
        addi    a4,s0,-576
        lw      a5,-24(s0)
        mv      a2,a3
        mv      a1,a4
        mv      a0,a5
        call    write@plt
        li      a0,3
        call    sleep@plt
        lbu     a5,-576(s0)
        mv      a4,a5
        li      a5,46
        beq     a4,a5,.L7
        j       .L2
.L7:
        nop
        lw      a5,-24(s0)
        mv      a0,a5
        call    close@plt
        li      a5,0
        mv      a0,a5
        ld      ra,568(sp)
        ld      s0,560(sp)
        addi    sp,sp,576
        jr      ra
```

To do:

1. Analyze the above code.
2. Modify the IP address and port number.
3. Execute together with the **server** (receiver) code on the same and on separate hosts
4. **Modify the stack use by implementing the buffer in global region.**

2.3.2.2 The receiver – server code with `listen()` and `accept()`

```
// tcprecv.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

#define BUFLEN 512      // Max length of buffer
#define PORT 8899       // The port on which to listen for connection

int main(void)
{
  struct sockaddr_in si_me, si_other;
  int s,ns, nb, slen = sizeof(si_other);
  char buf[BUFLEN];
  s=socket(AF_INET, SOCK_STREAM, 0);
  memset((char *) &si_other, 0, sizeof(si_other));
  si_me.sin_family = AF_INET;
  si_me.sin_port = htons(PORT);
  si_me.sin_addr.s_addr = htonl(INADDR_ANY);
  // Bind is mandatory for the server
  bind(s,(struct sockaddr *)&si_me , sizeof(si_me));
  listen(s,5);   // log length
  while(1)
    {
    //Accept and incoming connection
    puts("Waiting for incoming connections...");
    ns=accept(s, (struct sockaddr *)&si_other,(socklen_t*)&slen);
    if (ns<0) { puts("accept failed"); continue; }
    else  puts("Connection accepted");
    while(1)
      {
      memset(buf,0x00,BUFLEN);
      nb=read(ns,buf,BUFLEN);
      printf("Got message %d bytes: %s\n",nb,buf); if(nb==0) break;
      if(buf[0]=='.') { printf("session ended\n");break;}
      }
    close(ns);  // close the working socket
    }
  close(s);
}
```

And the corresponding assembly code:

```
       .file   "Recv.TCP.c"
        .option pic
        .attribute arch, "rv64i2p1_m2p0_a2p1_f2p2_d2p2_c2p0_zicsr2p0_zifencei2p0"
        .attribute unaligned_access, 0
        .attribute stack_align, 16
        .text
        .section        .rodata
        .align  3
.LC0:
        .string "Waiting for incoming connections..."
        .align  3
```

```
.LC1:
        .string "accept failed"
        .align  3
.LC2:
        .string "Connection accepted"
        .align  3
.LC3:
        .string "Got message %d bytes: %s\n"
        .align  3
.LC4:
        .string "session ended"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
        addi    sp,sp,-592
        sd      ra,584(sp)
        sd      s0,576(sp)
        addi    s0,sp,592
        li      a5,16
        sw      a5,-68(s0)
        li      a2,0
        li      a1,1
        li      a0,2
        call    socket@plt
        mv      a5,a0
        sw      a5,-20(s0)
        addi    a5,s0,-64
        li      a2,16
        li      a1,0
        mv      a0,a5
        call    memset@plt
       call    memset@plt
        li      a5,2
        sh      a5,-48(s0)
        li      a5,8192
        addi    a0,a5,707
        call    htons@plt
        mv      a5,a0
        sh      a5,-46(s0)
        li      a0,0
        call    htonl@plt
        mv      a5,a0
        sext.w  a5,a5
        sw      a5,-44(s0)
        addi    a4,s0,-48
        lw      a5,-20(s0)
        li      a2,16
        mv      a1,a4
        mv      a0,a5
        call    bind@plt
        lw      a5,-20(s0)
        li      a1,5
        mv      a0,a5
        call    listen@plt
.L8:
        lla     a0,.LC0
        call    puts@plt
        addi    a3,s0,-68
        addi    a4,s0,-64
        lw      a5,-20(s0)
        mv      a2,a3
        mv      a1,a4
        mv      a0,a5
        call    accept@plt
        mv      a5,a0
        sw      a5,-24(s0)
        lw      a5,-24(s0)
        sext.w  a5,a5
        bge     a5,zero,.L2
        lla     a0,.LC1
        call    puts@plt
        j       .L3
.L2:
        lla     a0,.LC2
        call    puts@plt
.L7:
        addi    a5,s0,-584
        li      a2,512
        li      a1,0
        mv      a0,a5
        call    memset@plt
        addi    a4,s0,-584
        lw      a5,-24(s0)
        li      a2,512
        mv      a1,a4
        mv      a0,a5
        call    read@plt
        mv      a5,a0
        sw      a5,-28(s0)
        addi    a4,s0,-584
        lw      a5,-28(s0)
        mv      a2,a4
        mv      a1,a5
        lla     a0,.LC3
        call    printf@plt
        lw      a5,-28(s0)
        sext.w  a5,a5
        beq     a5,zero,.L9
        lbu     a5,-584(s0)
        mv      a4,a5
        li      a5,46
        bne     a4,a5,.L7
        lla     a0,.LC4
        call    puts@plt
        j       .L5
.L9:
        nop
.L5:
        lw      a5,-24(s0)
        mv      a0,a5
        call    close@plt
.L3:
        j       .L8
```

To do:

1. Analyze the above code.
2. Modify the IP address and port number.
3. Execute together with the **server** (receiver) code on the same and on separate hosts
4. **Modify the stack use by implementing the buffer in global region.**

Lab 3

Designing fragmentary RISC-V with R-type instructions

In this lab we are going to study and experiment with a simple RISC-V model that simulates R-type instructions.
The model is prepared in Verilog language.
To start with, we remind you the basic elements of Verilog.

Verilog compiler and simulator may be installed with (Debian, Ubuntu) on any platform including RISC-V:

```
sudo apt install iverilog
```

To complete it with the waveform viewer :

```
sudo apt install gtkwave
```

3.1 Simple counter

The Verilog code -model:

```
module counter(out, clk, reset);
  parameter WIDTH = 8;
  output [WIDTH-1: 0] out;
  input           clk, reset;
  reg [WIDTH-1: 0]   out;
  wire           clk, reset;
  always @(posedge clk)
    out <= out + 1;
  always @reset
    if (reset)
      assign out = 0;
    else
      deassign out;
endmodule // counter
```

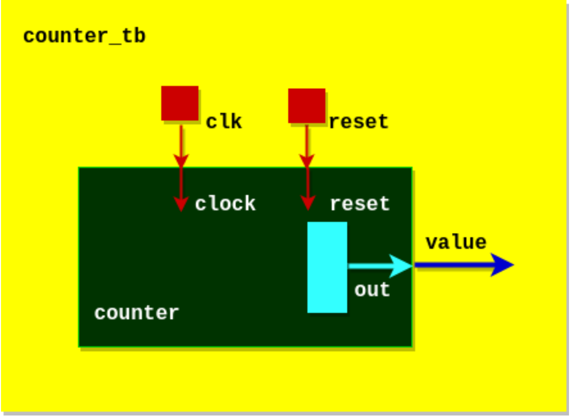


Figure 3.1 `counter` module and its test bench – `counter.tb`

The module has two input signals - **`wire`**: **`clk, reset`** and one registered – **`reg`** output: **`out[WIDTH-1:0]`**; where **`WIDTH`** is parameter.
Inside the module we have two permanent processes: **`always@`**, activated by positive clock edge @ **`(posedge clk)`** , and reset state high **`@reset`**.
The internal register out is connected permanently to the output wire. It operates as a M-S register with the positive edge for writing and always available output. It can be read and written in the same clock cycle:
`out <= out +1;`
The process with **`reset`** signal at high level **assigns** `0` to the output : **`out`**; otherwise it is deassigned (deconnected) from the **`out`** signal.

The Verilog code - model test bench:

To test the module counter we need the external stimuli and monitor commands. They are designed with via the test bench module in **`counter_tb.v file.`**

```
module counter_tb;
  /* Make a reset that pulses once. */
  reg reset = 0;
  initial begin
     $dumpfile("counter_wave.vcd");  // trace file
     $dumpvars;               // variables to dumb - all
     # 17 reset = 1;          // reset sequence
     # 11 reset = 0;
     # 29 reset = 1;
     # 5  reset =0;
     # 513 $finish;           // end of simulation run
  end
  reg clk = 0;
  always #1 clk =!clk;        // continuous process : clock wave
  wire [7:0] value;           // the value of counter
  counter c1 (value, clk, reset);   // here we instantiate the counter
  initial
     $monitor("At time %t,value =%h (%0d)",$time,value,value);

endmodule // counter_tb
```

The compilation:

```
bako@lpi4a:~/Programming/lab3$ iverilog counter_tb.v counter.v -o
counter_model
bako@lpi4a:~/Programming/lab3$ ls
counter.v  counter_model  counter_tb.v
bako@lpi4a:~/Programming/lab3$ ls -l
total 12
-rw-r--r-- 1 bako bako  318 Nov  3 16:15 counter.v
-rwxr-xr-x 1 bako bako 2823 Nov  3 16:19 counter_model
-rw-r--r-- 1 bako bako  622 Nov  3 16:16 counter_tb.v
bako@lpi4a:~/Programming/lab3$
```

The simulation:

```
vvp counter_model
```

```
bako@lpi4a:~/Programming/lab3$ vvp counter_model
VCD info: dumpfile test.vcd opened for output.
At time                   0,value =xx (x)
At time                  17,value =00 (0)
At time                  29,value =01 (1)
At time                  31,value =02 (2)
At time                  33,value =03 (3)
..
..
At time                  47,value =0a (10)
At time                  49,value =0b (11)
At time                  51,value =0c (12)
At time                  53,value =0d (13)
At time                  55,value =0e (14)
At time                  57,value =00 (0)
At time                  63,value =01 (1)
At time                  65,value =02 (2)

.. up to time 513 !
```

At the same time the simulator generated counter_wave.vcd file to be visioned via gtkwave.

```
bako@lpi4a:~/Programming/lab3$ ls -l
total 24
-rw-r--r-- 1 bako bako   318 Nov  3 16:15 counter.v
-rwxr-xr-x 1 bako bako  2831 Nov  3 16:25 counter_model
-rw-r--r-- 1 bako bako   630 Nov  3 16:25 counter_tb.v
-rw-r--r-- 1 bako bako 10747 Nov  3 16:25 counter_wave.vcd
bako@lpi4a:~/Programming/lab3$ gtkwave counter_wave.vcd
```

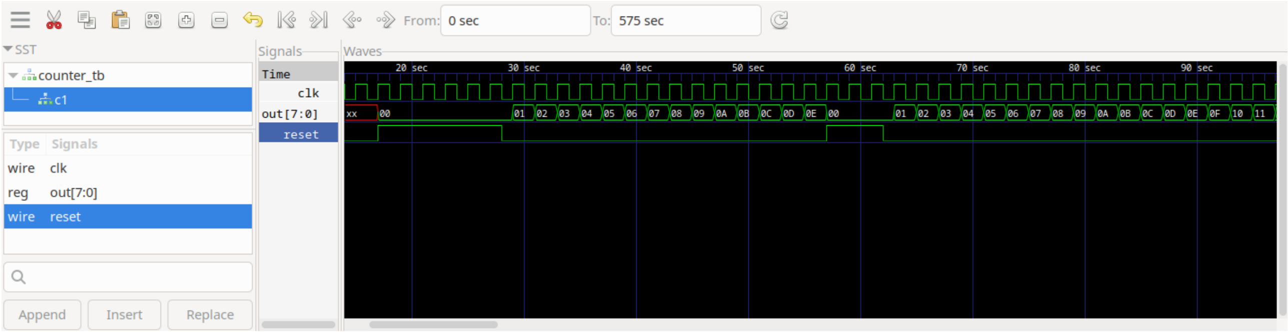


Figure 3.2 GTKwave output for `counter_wave.vcd` file

3.2 RAM memory block with two data busses

The memory block presented in this example has two parameters: **`AddressSize`** and **`WordSize`**.
The declaration:

```
        reg [WordSize-1:0] Mem [0:(1<<AddressSize)-1];
```

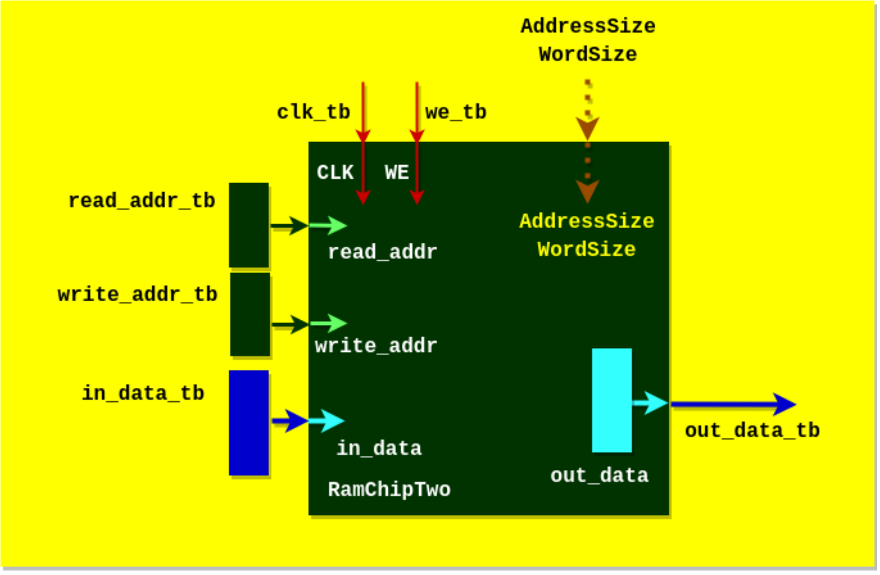


Figure 3.3 **`RamChipTwo`** module and its test-bench – **`RamChipTwo_tb`.**

Note that we use of the << operator for the 2^N implementation.
The **bidirectional port** is activated by a **conditional assignment**. This assignment actually models the process of reading from RAM. The **first `always`** block models the **writing process**. The second **`always`** performs a **simultaneous read-write condition**.
Note that with Verilog you need a temporary register to access a binary element of a memory word.

```
module RamChipTwo(read_addr,write_addr,in_data,out_data,CLK,WE);
parameter AddressSize=1;
parameter WordSize=1;
input [AddressSize-1:0]read_addr,write_addr;
input [WordSize-1:0]in_data;
output reg [WordSize-1:0]out_data;
input CLK,WE;
        // Declare the RAM variable
        reg [WordSize-1:0] Mem[0:(1<<AddressSize)-1];
        always @ (posedge CLK)
        begin
                // Write
                if (WE)
                        Mem[write_addr] <= in_data;
                out_data <= Mem[read_addr];
        end
endmodule
```

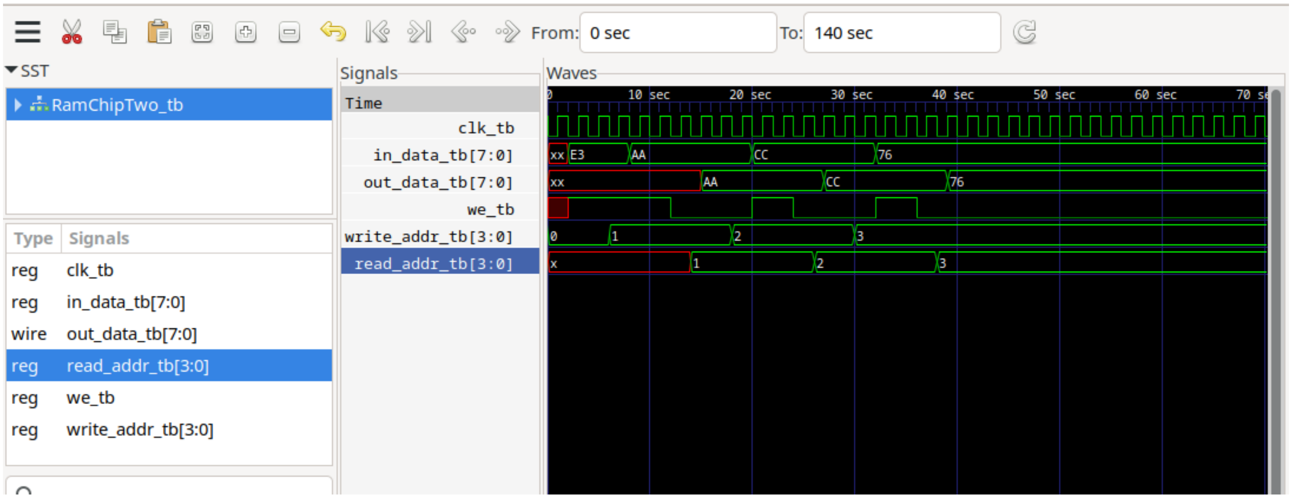
Here is the test-bench module:

```
module RamChipTwo_tb ();
reg clk_tb =0;
reg[3:0] read_addr_tb, write_addr_tb;
wire[7:0] out_data_tb;
reg[7:0] in_data_tb;

defparam mem0.AddressSize=4;
defparam mem0.WordSize=8;
RamChipTwo mem0(read_addr_tb,write_addr_tb,in_data_tb,out_data_tb,clk_tb,we_tb);

initial begin
```

```
  $dumpfile("RamChipTwo_wave.vcd");
  $dumpvars;
  write_addr_tb = 4'b0000;
  write_addr_tb = 4'b0000;
  #2 we_tb =1; in_data_tb=8'b11100011;
  #2 $display("write_data=%b, read_data=%b",in_data_tb,out_data_tb);
  #2 write_addr_tb =4'b0001;
  #2 we_tb =1; in_data_tb=8'b10101010;
  #2 $display("write_data=%b, read_data=%b",in_data_tb,out_data_tb);
  #2 we_tb=0;
  #2 read_addr_tb = 4'b0001;
  #2 $display("write_data=%b, read_data=%b",in_data_tb,out_data_tb);
  #2 write_addr_tb = 4'b0010;
  #2 we_tb =1; in_data_tb=8'b11001100;
  #2 $display("write_data=%b, read_data=%b",in_data_tb,out_data_tb);
  #2 we_tb=0;
  #2 read_addr_tb = 4'b0010;
  #2 $display("write_data=%b, read_data=%b",in_data_tb,out_data_tb);
  #2 write_addr_tb = 4'b0011;
  #2 we_tb =1; in_data_tb=8'b1110110;
  #2 $display("write_data=%b, read_data=%b",in_data_tb,out_data_tb);
  #2 we_tb=0;
  #2 read_addr_tb = 4'b0011;
  #2 $display("write_data=%b, read_data=%b",in_data_tb,out_data_tb);
  #100 $finish;
end

-------------------
sipeed@lpi4a:~/Programming/lab3$ iverilog RamChipTwo_tb.v RamChipTwo.v -o RamChipTwo_model
sipeed@lpi4a:~/Programming/lab3$ vvp RamChipTwo_model
VCD info: dumpfile RamChipTwo_wave.vcd opened for output.
write_data=11100011, read_data=xxxxxxxx
write_data=10101010, read_data=xxxxxxxx
write_data=10101010, read_data=10101010
write_data=11001100, read_data=10101010
write_data=11001100, read_data=11001100
write_data=01110110, read_data=11001100
write_data=01110110, read_data=01110110
sipeed@lpi4a:~/Programming/lab3$
```

The generated waveform.

```
sipeed@lpi4a:~/Programming/lab3$ gtkwave RamChipTwo_wave.vcd
GTKWave Analyzer v3.3.115 (w)1999-2023 BSI
[0] start time.
[140] end time.
```



Type	Signals
reg	clk_tb
reg	in_data_tb[7:0]
wire	out_data_tb[7:0]
reg	read_addr_tb[3:0]
reg	we_tb
reg	write_addr_tb[3:0]

Figure 3.4 GTKWave form for **RamChipTwo_tb** module simulation.

3.3 Simple RISC-V Verilog model for R-type instructions

Most of modern implementations of RISC-V are developed with Verilog hardware description and design language. In our case we will try to model a limited part of RISC-V with **R-type** instructions only.

31 25	24 20	19 15	14 12	11 7	6 0
funct7	rs2	rs1	funct3	rd	opcode
7	5	5	3	5	7
0000000	src2	src1	ADD/SLT/SLTU	dest	OP
0000000	src2	src1	AND/OR/XOR	dest	OP
0000000	src2	src1	SLL/SRL	dest	OP
0100000	src2	src1	SUB/SRA	dest	OP

Figure 3.5 R_type instructions format with 3 register address fields: `src1`, `src2`, and `sd`.

Figure 3.6 shows the required components to execute **R-Type** instructions (**Register-To-Register**).

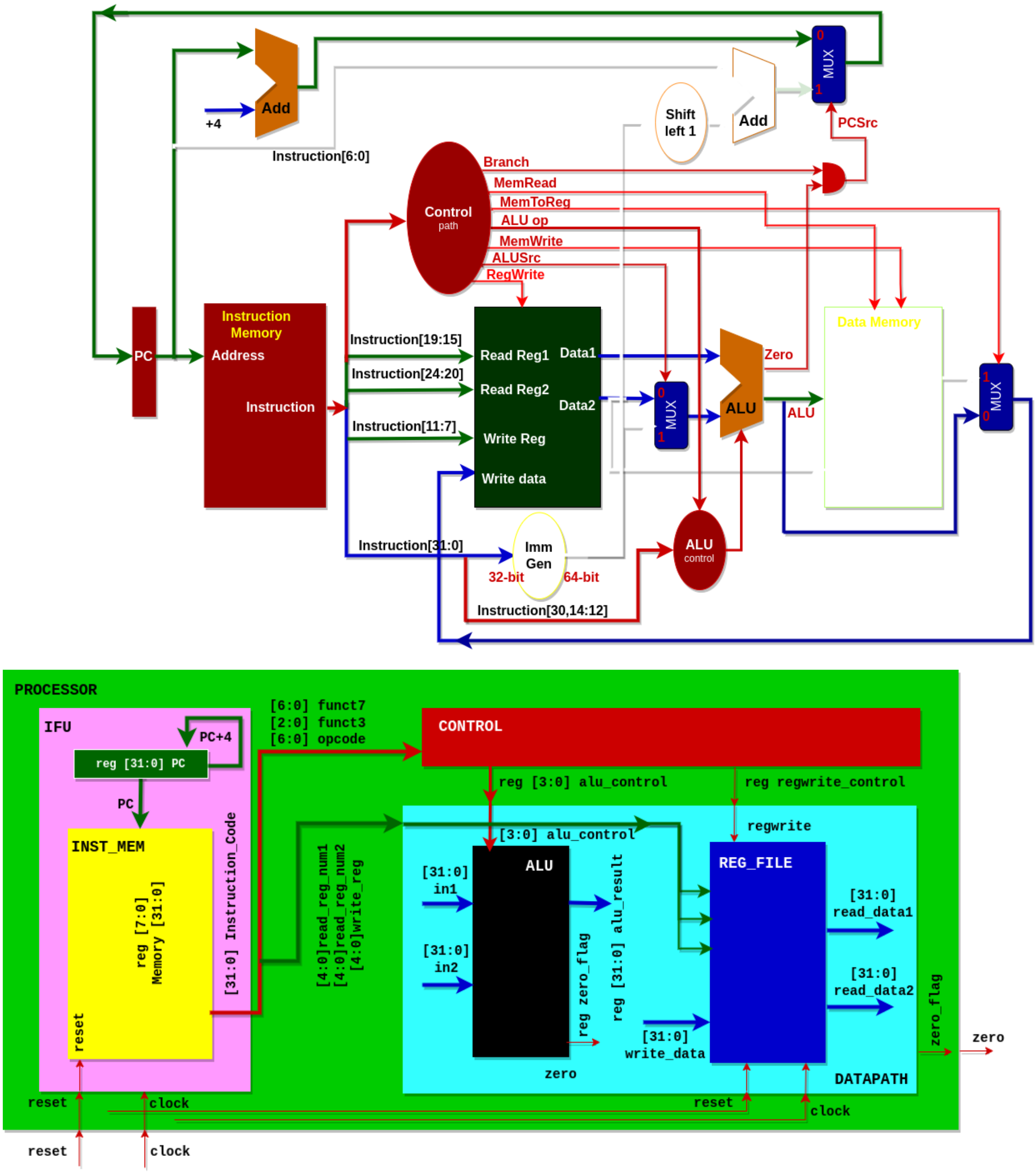


Figure 3.6 Datapath and control lines for the execution of R-type instructions and corresponding Verilog model

Let us present the modules illustrated in **Figure 3.6.**

The top level **module** of the Verilog code is called `PROCESSOR.v` and it looks like this:

```
`include "CONTROL.v"
`include "DATAPATH.v"
`include "IFU.v"

module PROCESSOR(
    input clock,
    input reset,
    output zero
);

    wire [31:0] instruction_code;
    wire [3:0] alu_control;
    wire regwrite;

    IFU IFU_module(clock, reset, instruction_code);

    CONTROL control_module(instruction_code[31:25],instruction_code[14:12],instruction_code[6:0],
                           alu_control, regwrite);

    DATAPATH datapath_module(instruction_code[19:15], instruction_code[24:20],
                             instruction_code[11:7], alu_control, regwrite, clock, reset, zero);

endmodule
```

The `PROCESSOR` module includes the control module `CONTROL.v`, the datapath module `DATAPATH.v` and the instruction fetch module `IFU.v` .
The module has only **two simple signal inputs**: `clock` and `reset`, and **one output: `zero`.**
The internal connections are wires to carry the `instruction_code` – **32 bits**, the `alu_control` – **4 bits**, and simple control signal `regwrite`.

The module instantiates 3 sub-modules:

```
IFU IFU_module(clock, reset, instruction_code);
```

that receives clock and reset signals and sends one output signal `instruction_code` on 32 bits.

```
CONTROL control_module(instruction_code[31:25],
                       instruction_code[14:12],instruction_code[6:0],
                       alu_control, regwrite);
```

That receives the instruction code with `funct7 – [31:25]`, `funct3 – [14:12]` and `opcode` fields – `[6:0]`; and sends `alu_control` signal on 4 bits, and simple `regwrite` signal.
Inside the `CONTROL` module , the output signals are registered as:

```
    output reg [3:0] alu_control,
    output reg regwrite_control
```

```
DATAPATH datapath_module(instruction_code[19:15], instruction_code[24:20],
                         instruction_code[11:7], alu_control,
                         regwrite, clock, reset, zero);
```

`DATAPATH` module receives the **addresses of 3 registers**, two input registers: `[11:7]`, `[24:20]` and one output register `[19:15]`. The module also takes **seven control signals**: four at `alu_control` and one at `regwrite`, `clock`, `reset`. The module outputs **one signal**: `zero`.

3.3.1 Control path

Inside the **`IFU`** (**Instruction Fetch Unit**) we find the program counter : **`reg [31:0] PC`** ; and the **`INST_MEM`** submodule.

3.1.1.1 Instruction fetch unit - `IFU`

The following is the Verilog code of **`IFU`** module:

```
`include "INST_MEM.v"

module IFU(
    input clock,reset,
    output [31:0] Instruction_Code
);
reg [31:0] PC = 32'b0;  // 32-bit program counter is initialized to zero

    // Initializing the instruction memory block
    INST_MEM instr_mem(PC,reset,Instruction_Code);

    always @(posedge clock, posedge reset)
    begin
        if(reset == 1)  //If reset is one, clear the program counter
        PC <= 0;
        else
        PC <= PC+4;   // Increment program counter on positive clock edge
    end

endmodule
```

The instruction fetch unit has **`clock`** and **`reset`** pins as input and **32-bit instruction code as output**. Internally the block has Instruction Memory (**`INST_MEM instr_mem`**), Program Counter (**`PC`**) and an adder to **increment counter by 4**, on **every positive clock edge**.

```
        always @(posedge clock, posedge reset)
```

3.3.1.2 Instruction Memory unit - `INST_MEM`

`INST_MEM` module has two inputs: address line from **`PC`** on 32 bits, and **`reset`** signal. It has one output on 32-bits, that carries **`Instruction_Code`**.

`INST_MEM` module integrates a modest size of 128 bytes organized in 32 4-byte words.

```
    reg [7:0] Memory [31:0]; // Byte addressable memory with 32 locations

module INST_MEM(
    input [31:0] PC,
    input reset,
    output [31:0] Instruction_Code
);
    reg [7:0] Memory [31:0]; // Byte addressable memory with 32 locations

    // Under normal operation (reset = 0), we assign the instr. code, based on PC
    assign Instruction_Code = {Memory[PC+3],Memory[PC+2],Memory[PC+1],Memory[PC]};

    // Initializing memory when reset is one
    always @(reset)
    begin
        if(reset == 1)
        begin
            // Setting 32-bit instruction: add t1, s0,s1 => 0x00940333
            Memory[3] = 8'h00;
            Memory[2] = 8'h94;
            Memory[1] = 8'h03;
            Memory[0] = 8'h33;
            // Setting 32-bit instruction: sub t2, s2, s3 => 0x413903b3
            Memory[7] = 8'h41;
            Memory[6] = 8'h39;
            Memory[5] = 8'h03;
            Memory[4] = 8'hb3;
            // Setting 32-bit instruction: mul t0, s4, s5 => 0x035a02b3
            Memory[11] = 8'h03;
            Memory[10] = 8'h5a;
            Memory[9] = 8'h02;
            Memory[8] = 8'hb3;
            // Setting 32-bit instruction: xor t3, s6, s7 => 0x017b4e33
            Memory[15] = 8'h01;
            Memory[14] = 8'h7b;
            Memory[13] = 8'h4e;
            Memory[12] = 8'h33;
            // Setting 32-bit instruction: sll t4, s8, s9
            Memory[19] = 8'h01;
            Memory[18] = 8'h9c;
            Memory[17] = 8'h1e;
            Memory[16] = 8'hb3;
            // Setting 32-bit instruction: srl t5, s10, s11
            Memory[23] = 8'h01;
            Memory[22] = 8'hbd;
            Memory[21] = 8'h5f;
            Memory[20] = 8'h33;
            // Setting 32-bit instruction: and t6, a2, a3
            Memory[27] = 8'h00;
            Memory[26] = 8'hd6;
            Memory[25] = 8'h7f;
            Memory[24] = 8'hb3;
            // Setting 32-bit instruction: or a7, a4, a5
            Memory[31] = 8'h00;
            Memory[30] = 8'hf7;
            Memory[29] = 8'h68;
            Memory[28] = 8'hb3;
        end
    end

endmodule
```

Instruction memory takes in two inputs: a **32-bit Program Counter** and a 1-bit **`reset`**.
The memory is **initialized** when **`reset`** is **`1`**. When reset is set to 0, Based on the value of **`PC`**, corresponding 32-bit Instruction code is output.

3.3.2 Data path

`DATAPATH` module is built from two submodules: **`REG_FILE`** module and **`ALU`** module. The register address fields - signals are connected to **`REG_FILE`** address inputs to select the required registers (two input registers, and one output register). The **`REG_FILE`** has 3 data busses to carry the data from/to selected registers, as well as 3 control signals for: **`regwrite`**, **`clock`**, and **`reset`**.

`ALU` module inputs are: **`[31:0] in1,in2`** for input data on 32-bit bus, and the output result on **`reg [31:0] alu_result.`** Note that the output is **registered.** **`ALU`** control is provided via **`[3:0] alu_control`** lines. The second “result” - output from the ALU module is **`zero_flag`** signal.

3.3.2.1 Register file : 32 32-bit registers

A register file can read two registers and write in to one register at the “same” time. The RISC V register file contains total of 32 registers each of size 32-bit. Hence 5-bits are used to specify the register numbers that are to be read or written.
Register file **always outputs** the contents of the register corresponding to read register numbers specified. Reading a register **is not dependent** on any other signals.
Register writes are controlled by a control signal **`RegWrite.`** The write should happen if **`RegWrite`** signal is set to 1 and if there is **positive edge of clock**.

```
module REG_FILE(
    input [4:0] read_reg_num1,
    input [4:0] read_reg_num2,
    input [4:0] write_reg,
    input [31:0] write_data,
    output [31:0] read_data1,
    output [31:0] read_data2,
    input regwrite,
    input clock,
    input reset
);

    reg [31:0] reg_memory [31:0]; // 32 memory locations each 32 bits wide
    integer i=0;
    //  When reset is triggered, we initialize the registers with some values
    always @(posedge reset)
    begin
         reg_memory[0] = 32'h0;
         reg_memory[1] = 32'h1;
         reg_memory[2] = 32'h2;
         reg_memory[3] = 32'h3;
         reg_memory[4] = 32'h4;
         reg_memory[5] = 32'h5;
         reg_memory[6] = 32'h6;
         reg_memory[7] = 32'h7;
         reg_memory[8] = 32'h8;
         reg_memory[9] = 32'h9;
         reg_memory[10] = 32'h10;
         reg_memory[11] = 32'h11;
         reg_memory[12] = 32'h12;
         reg_memory[13] = 32'h13;
         reg_memory[14] = 32'h14;
         reg_memory[15] = 32'h15;
         reg_memory[16] = 32'h16;
         reg_memory[17] = 32'h17;
         reg_memory[18] = 32'h18;
         reg_memory[19] = 32'h19;
         reg_memory[20] = 32'h20;
         reg_memory[21] = 32'h21;
         reg_memory[22] = 32'h22;
         reg_memory[23] = 32'h23;
         reg_memory[24] = 32'h24;
         reg_memory[25] = 32'h25;
         reg_memory[26] = 32'h26;
         reg_memory[27] = 32'h27;
         reg_memory[28] = 32'h28;
         reg_memory[29] = 32'h29;
         reg_memory[30] = 32'h30;
         reg_memory[31] = 32'h31;

    end

    assign read_data1 = reg_memory[read_reg_num1];
    assign read_data2 = reg_memory[read_reg_num2];

    // If clock edge is positive and regwrite is 1, we write data to specified register
    always @(posedge clock)
    begin
        if (regwrite) begin
            reg_memory[write_reg] = write_data;
        end
    end

endmodule
```

3.3.2.2 ALU unit

ALU module takes two operands of size 32-bits each and a 4-bit **alu_control** as input. Operation is performed on the basis of **alu_control** value and output is 32-bit **ALU_result**. If the **alu_result** is zero, a **zero_flag** is set.

ALU Control lines (**alu_control**) operation:

0000	**Bitwise-AND**
0001	**Bitwise-OR**
0010	**Add (A+B)**
0100	**Subtract (A-B)**
1000	**Set on less than**
0011	**Shift left logical**
0101	**Shift right logical**
0110	**Multiply**
0111	**Bitwise-XOR**

```
module ALU (
    input [31:0] in1,in2,
    input[3:0] alu_control,
    output reg [31:0] alu_result,
    output reg zero_flag
);
    always @(*)
    begin
        // Operating based on control input
        case(alu_control)

        4'b0000: alu_result = in1&in2;
        4'b0001: alu_result = in1|in2;
        4'b0010: alu_result = in1+in2;
        4'b0100: alu_result = in1-in2;
        4'b1000: begin
            if(in1<in2)
            alu_result = 1;
            else
            alu_result = 0;
        end
        4'b0011: alu_result = in1<<in2;
        4'b0101: alu_result = in1>>in2;
        4'b0110: alu_result = in1*in2;
        4'b0111: alu_result = in1^in2;

        endcase

        // Setting Zero_flag if ALU_result is zero
        if (alu_result == 0)
            zero_flag = 1'b1;
        else
            zero_flag = 1'b0;

    end
endmodule
```

3.3.3 Processor test

The test of the **PROCESSOR** module requires the preparation of test bench module. The test bench module integrates the **PROCESSOR** module and provides the stimuli to animate it.
The stimuli signals are mainly reset and clock.

Initially **reset** prepares the starting time (here **#50** time units – nanoseconds). The second **initial** block starts the generation of the clock signal that has 2*20 nanosecond cycle (**#20 clock = ~clock**)and turns **forever.**

Note that the initial **initial** block prepares the name of the **trace file** - **dumpfile** and indicates the names off variables/signals to be registered. In case of first argument = **0,** all variables are to be traced.

```
`include "PROCESSOR.v"

module stimulus ();

    reg clock;
    reg reset;
    wire zero;

    // Instantiating the processor!!!
    PROCESSOR test_processor(clock,reset,zero);

    initial begin
        $dumpfile("output_wave.vcd");
        $dumpvars(0,stimulus);
    end

    initial begin
        reset = 1;
        #50 reset = 0;
    end
    initial begin
        clock = 0;
        forever #20 clock = ~clock;
    end

    initial
    #300 $finish;

endmodule
```

Let us call register the above code in `Processor_tb.v` file.

Now we have all components to start the **compilation** and **simulation of the compiled model**.

If you have previously installed the (Icarus Verilog – **iverilog**, and **GTKWave** tool) compiler/simulator you can **compile** the model (having all modules in the same directory) with:

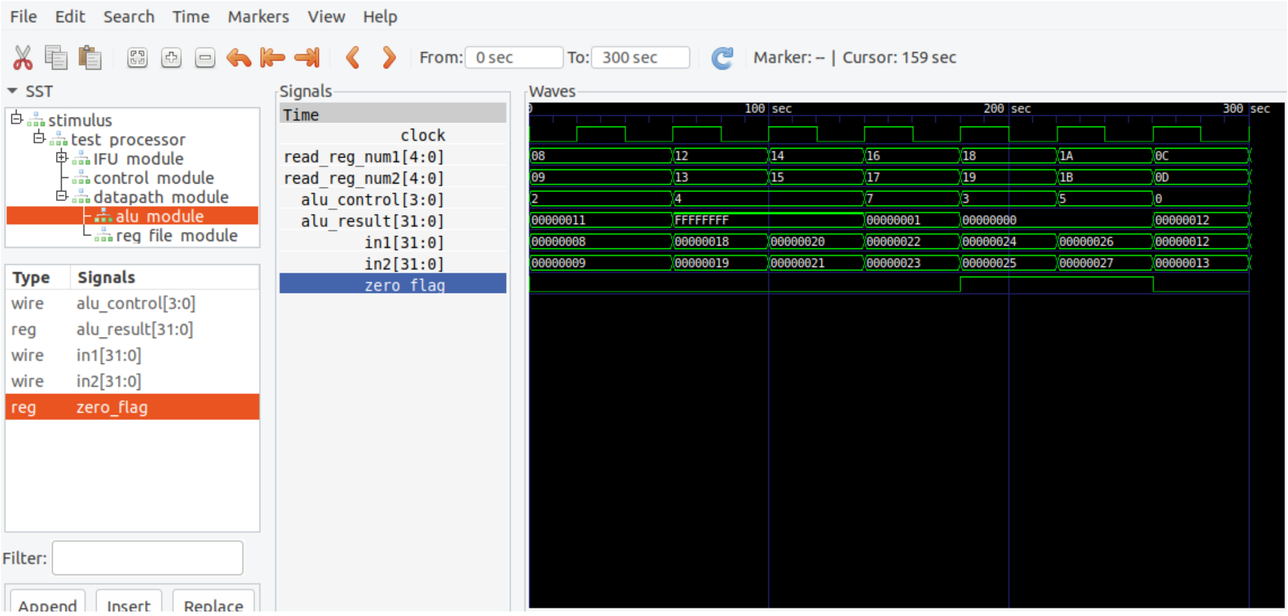
```
iverilog Processor_tb.v  -o  proc-compiled
```

After the compilation you can **execute** the compiled model with:

```
vvp proc-compiled
```

The waveform generated from **testbench** is named as `output_wave.vcd`

```
gtkwave output_wave.vcd
```



Type	Signals
wire	alu_control[3:0]
reg	alu_result[31:0]
wire	in1[31:0]
wire	in2[31:0]
reg	zero_flag

```
clock
read_reg_num1[4:0]   08        12        14        16        18        1A        0C
read_reg_num2[4:0]   09        13        15        17        19        1B        0D
alu_control[3:0]     2         4                   7         3         5         0
alu_result[31:0]     00000011  FFFFFFFF            00000001  00000000            00000012
in1[31:0]            00000008  00000018  00000020  00000022  00000024  00000026  00000012
in2[31:0]            00000009  00000019  00000021  00000023  00000025  00000027  00000013
zero_flag
```

Figure 3.7 GTKWave form for the given program simulation

You can trace the `alu_result` for operation `alu_oper = 4` meaning **subtraction** (look in the module `ALU`) and two input values: `in1=18`, `in2=19`, **what is the result ?**

To do:

1. Analyze the “given” result
2. Modify the Instruction Memory content using “your” program – instruction sequence and test it.

Lab 4

Designing complete RISC-V (RV32I)

The following lab is based on the complete `RV32I` model including all types of Integer instructions. The code developed as a single module with several fragments:

- interface
- decoder
- register file
- ALU unit
- address generation
- write-back
- load and store
- state machine

The interface part defines the input and output signals (wires) and provides some parameters that are also fixed locally.

```
wire[7:0] funct3Is=8'b00000001 << instr[14:12];
```

generates one hot value of the op-code for the decoder fragment.

In the decoder fragment we prepare a set of immediate values depending on the type of instruction. For example the

```
Iimm = {{21{instr[31]}}, instr[30:20]};
```

generates the signal with 21 replications of `instr[31]` value and 11 bits from `instr[30:20]` field.

The decoder initializes the instruction type signals:

```
isLoad, isALUimm,.., isJAL, isALU
```

to be used in the ALU and control loop.

The register file is declared as:

```
reg[31:0] registerFile[31:0];
```

there are also two source registers:

```
reg[31:0] rs1,rs2
```

The process:

```
always@(posedge clk)
  if(writeBack&&rdId!=0) registerFile[rdId]<=writeBackData;
```

writes (back) the data to register file with `rdId` identifier that must be different to zero.

In the ALU fragment the unique assignment provides the selection and operations of the ALU:

```
wire[31:0] aluOut =
  (funct3Is[0] ? instr[30] & instr[5] ? aluMinus[31:0] : aluPlus : 32'b0) |
  (funct3Is[2] ? {31'b0,LT} : 32'b0)       |
  (funct3Is[3] ? {31'b0, LTU} : 32'b0)     |
  (funct3Is[4] ? aluIn1^aluIn2 : 32'b0)    |
  (funct3Is[6] ? aluIn1|aluIn2 : 32'b0)    |
  (funct3Is[7] ? aluIn1 & aluIn2 : 32'b0) |
  (funct3IsShift ? aluReg : 32'b0) ;
```

Note that the selection is done via `funct3Is[x]` – hot code and if the operation is not selected it generates logic value 0 (neutral to `| wide-or` operator) on all bits.

```
module miniRV32I(input clk,input mem_wbusy, input reset,
       output[31:0] mem_addr, output[31:0] mem_wdata,
       output[3:0] mem_wmask, input[31:0] mem_rdata,
       output mem_rstrb, input mem_rbusy);

       parameter RESET_ADDR = 32'h00000000;
       parameter ADDR_WIDTH = 24;
       localparam ADDR_PAD = {(32-ADDR_WIDTH){1'b0}};
       wire[4:0] rdId=instr[11:7];
       (* onehot *) wire[7:0] funct3Is=8'b00000001 << instr[14:12];

       /******************************* DECODER ***********************************/
       wire[31:0]  Uimm = {    instr[31],   instr[30:12], {12{1'b0}}};
       wire[31:0]  Iimm = {{21{instr[31]}}, instr[30:20]};
       wire[31:0]  Simm = {{21{instr[31]}}, instr[30:25],instr[11:7]};
       wire[31:0]  Bimm = {{20{instr[31]}}, instr[7],instr[30:25],instr[11:8],1'b0};
       wire[31:0]  Jimm = {{12{instr[31]}}, instr[19:12],instr[20],instr[30:21],1'b0};

       wire isLoad  =(instr[6:2] == 5'b00000); wire isALUimm=(instr[6:2] == 5'b00100);
       wire isAUIPC =(instr[6:2] == 5'b00101); wire isStore =(instr[6:2] == 5'b01000);
       wire isALUreg=(instr[6:2] == 5'b01100); wire isLUI   =(instr[6:2] == 5'b01101);
       wire isBranch=(instr[6:2] == 5'b11000); wire isJALR  =(instr[6:2] == 5'b11001);
       wire isJAL   =(instr[6:2] == 5'b11011); wire isALU   =isALUimm | isALUreg;

       /************************** REGISTER FILE *********************************/

       reg[31:0] rs1; reg[31:0] rs2; reg[31:0] registerFile[31:0];
       always@(posedge clk) if(writeBack&&rdId!=0) registerFile[rdId]<=writeBackData;

       /**************************  ALU  ****************************************/
       wire funct3IsShift = funct3Is[1] | funct3Is[5];
       wire[31:0] aluIn1 = rs1;
       wire[31:0] aluIn2 = isALUreg | isBranch ? rs2: Iimm;
       reg[31:0] aluReg;
       reg[4:0] aluShamt;
       wire aluBusy = | aluShamt;
       wire aluWr;
       wire[31:0] aluPlus = aluIn1 + aluIn2;
       wire[32:0] aluMinus = {1'b1,~aluIn2} + {1'b0,aluIn1} + 33'b1;
       wire LT  = (aluIn1[31] ^ aluIn2[31]) ? aluIn1[31] : aluMinus[32];
       wire LTU = aluMinus[32];
       wire EQ = (aluMinus[31:0] == 0);
       wire[31:0] aluOut =
         (funct3Is[0] ? instr[30] & instr[5] ? aluMinus[31:0] : aluPlus : 32'b0) |
         (funct3Is[2] ? {31'b0,LT} : 32'b0)      |
         (funct3Is[3] ? {31'b0, LTU} : 32'b0)    |
         (funct3Is[4] ? aluIn1^aluIn2 : 32'b0)   |
         (funct3Is[6] ? aluIn1|aluIn2 : 32'b0)   |
         (funct3Is[7] ? aluIn1 & aluIn2 : 32'b0) |
         (funct3IsShift ? aluReg : 32'b0) ;

       always@(posedge clk)
       begin
         if(aluWr)
              begin if (funct3IsShift)
                     begin
                     aluReg <= aluIn1; aluShamt <= aluIn2[4:0];
                     end
         end
         if(|aluShamt)
              begin aluShamt <= aluShamt-1;
              aluReg<=funct3Is[1]?aluReg<<1: {instr[30] & aluReg[31], aluReg[31:1]};
         end
       end
       wire predicate = funct3Is[0] & EQ | funct3Is[1] & !EQ | funct3Is[4] & LT |
              funct3Is[5] & !LT | funct3Is[6] & LTU | funct3Is[7] & !LTU;

       /********************  ADDRESS GENERATION *******************************/
       reg[ADDR_WIDTH-1:0] PC;
       reg[31:2] instr;
       wire [ADDR_WIDTH-1:0] PCplus4 = PC+4;
       wire[ADDR_WIDTH-1:0] PCplusImm = PC + (instr[3] ? Jimm[ADDR_WIDTH-1:0] :
              instr[4] ? Uimm[ADDR_WIDTH-1:0] : Bimm[ADDR_WIDTH-1:0]);
       wire[ADDR_WIDTH-1:0] loadstore_addr = rs1[ADDR_WIDTH-1:0] +
              (instr[5] ? Simm[ADDR_WIDTH-1:0] : Iimm[ADDR_WIDTH-1:0]);
```

```
        assign mem_addr = {ADDR_PAD, state[WAIT_INSTR_bit] | state[FETCH_INSTR_bit] ?
            PC : loadstore_addr};

        /**********************  WRITE BACK **************************************/
        wire[31:0] writeBackData = (isLUI ? Uimm : 32'b0) |
            (isALU ? aluOut : 32'b0) | (isAUIPC ? {ADDR_PAD,PCplusImm} : 32'b0) |
            (isJALR|isJAL ? {ADDR_PAD,PCplus4}:32'b0) | (isLoad ? LOAD_data: 32'b0);

        /**********************  LOAD:STORE  ************************************/
        wire mem_byteAccess     = instr[13:12] == 2'b00;
        wire mem_halfwordAccess = instr[13:12] == 2'b01;

        wire LOAD_sign=!instr[14] & (mem_byteAccess ? LOAD_byte[7] : LOAD_halfword[15]);
        wire[31:0] LOAD_data = mem_byteAccess ? {{24{LOAD_sign}},LOAD_byte} :
               mem_halfwordAccess ? {{16{LOAD_sign}}, LOAD_halfword} : mem_rdata ;

        wire[15:0] LOAD_halfword = loadstore_addr[1] ? mem_rdata[31:16]:mem_rdata[15:0];
        wire[7:0] LOAD_byte = loadstore_addr[0]? LOAD_halfword[15:8]:LOAD_halfword[7:0];

        assign mem_wdata[7:0]   = rs2[7:0];
        assign mem_wdata[15:8]  = loadstore_addr[0] ? rs2[7:0] : rs2[15:8];
        assign mem_wdata[23:16] = loadstore_addr[1] ? rs2[7:0] : rs2[23:16];
        assign mem_wdata[31:24] = loadstore_addr[0] ? rs2[7:0] :
                                  loadstore_addr[1] ? rs2[15:8]: rs2[31:24];
        wire[3:0] STORE_wmask = mem_byteAccess ? (loadstore_addr[1] ?
               (loadstore_addr[0]?4'b1000:4'b0100) : (loadstore_addr[0] ? 4'b0010:4'b0001)) :
               mem_halfwordAccess ? (loadstore_addr[1] ? 4'b1100 : 4'b0011) : 4'b1111;

        /*********************  STATE MACHINE ************************************/
        localparam FETCH_INSTR_bit      = 0, FETCH_INSTR = 1 << FETCH_INSTR_bit;
        localparam WAIT_INSTR_bit       = 1, WAIT_INSTR  = 1 << WAIT_INSTR_bit;
        localparam EXECUTE_bit          = 2, EXECUTE     = 1 << EXECUTE_bit;
        localparam WAIT_ALU_OR_MEM_bit  = 3, WAIT_ALU_OR_MEM = 1 << WAIT_ALU_OR_MEM_bit;

        (* onehot *) reg[3:0] state;
        wire writeBack = ~(isBranch | isStore ) & (state[EXECUTE_bit] | state[WAIT_ALU_OR_MEM_bit]);

        assign mem_rstrb = state[EXECUTE_bit] & isLoad | state[FETCH_INSTR_bit];
        assign mem_wmask = {4{state[EXECUTE_bit] & isStore}} & STORE_wmask;
        assign aluWr = state[EXECUTE_bit] & isALU;
        wire jumpToPCplusImm = isJAL | (isBranch & predicate);
        wire needToWait = isLoad | isStore | isALU & funct3IsShift;

        always@(posedge clk)
          begin
          if(!reset)
               begin
                 state <= WAIT_ALU_OR_MEM; PC <= RESET_ADDR[ADDR_WIDTH-1:0];
               end
          else
            (* parallel_case *)
            case(1'b1)
               state[WAIT_INSTR_bit]:
                 begin
                 if(!mem_rbusy)
                       begin
                       rs1<=registerFile[mem_rdata[19:15]]; rs2<=registerFile[mem_rdata[24:20]];
                       instr <= mem_rdata[31:2]; state <= EXECUTE;
                 end
               end
               state[EXECUTE_bit]:
                 begin
                   PC<=isJALR?{aluPlus[ADDR_WIDTH-1:1],1'b0}:jumpToPCplusImm?PCplusImm:PCplus4;
                   state <= needToWait ? WAIT_ALU_OR_MEM : FETCH_INSTR;
                 end
               state[WAIT_ALU_OR_MEM_bit]:
                 begin
                 if(!aluBusy & !mem_rbusy & !mem_wbusy) state <= FETCH_INSTR;
                 end
               default: state<= WAIT_INSTR;
            endcase
          end
endmodule
```

Table of Contents