SMART-V Documentation

RI5CY-based Implementation with RTOS-tailored Memory Protection.
Still work-in-progress

Maja Malenko
malenko@student.tugraz.com
malenko.maya@gmail.com

January 2022

1 Overview

This *RISC-V* implementation is a modified version of the *RI5CY* core[1]. It has a 32-bit 4-stage single-issue pipeline with an in-order schedule.
Some modules from the original *RI5CY* implementation which are not important for my concept[2] are removed, e.g., hardware loops, post-incremental load and store, additional ALU instructions not part of the standard ISA, performance counters, debug unit, floating point unit, and tracer. Thus, only the basic RV32IM version is kept.
The PMP Unit, as implemented in the *RI5CY* project is completely removed, and is substituted with a new MPU unit, specifically tailored for my PhD Thesis. [MM: More description]

The purpose of this specification is only to show some basic Test Cases implemented completely in assembly and tackling some basic RISC-V design concepts. They are described in details in Section 3, along with the simulation waveforms and explanation of the most relevant pipeline signals. The description is aimed to help developers who have just started using a *RISC-V* core, and particularly the *RI5CY* implementation (including the CV32E40P CORE-V from OpenHW), to better understand its internals.

Note:

---

2 Toolchain

This section discusses the prerequisites for compiling, linking, and simulating the Test Cases from Section 3.

2.1 Makefile

Makefile for the test cases:

Makefile for the core:
Analyze/Compile: **xvlog** and **xvhdl** parse Verilog and VHDL files and store the parsed files into an HDL lib. *in xsim.dir/work/*.sdb*
Elaborate: **xelab** loads all sub-design units for a given top-level unit, performs static elaboration, and linkes the generated executable code with the simulation kernel to create an executable simulation snapshot. *in xsim.dir/snapshot ... xsimk*
Simulate: xsim loads a simulation snapshot in an interactive simulation environment.

In order to have a meaningful description of the registers, the testbench instantiates all 32 of them.

[1]github
[2]my phd

2.2 Linker Script

The Linker Script (link.ld) is set in accordance to our OS memory model as well as the memory controller (memory_controller.sv). Both can be changed accordingly.

Listing 1: Example Linker Script

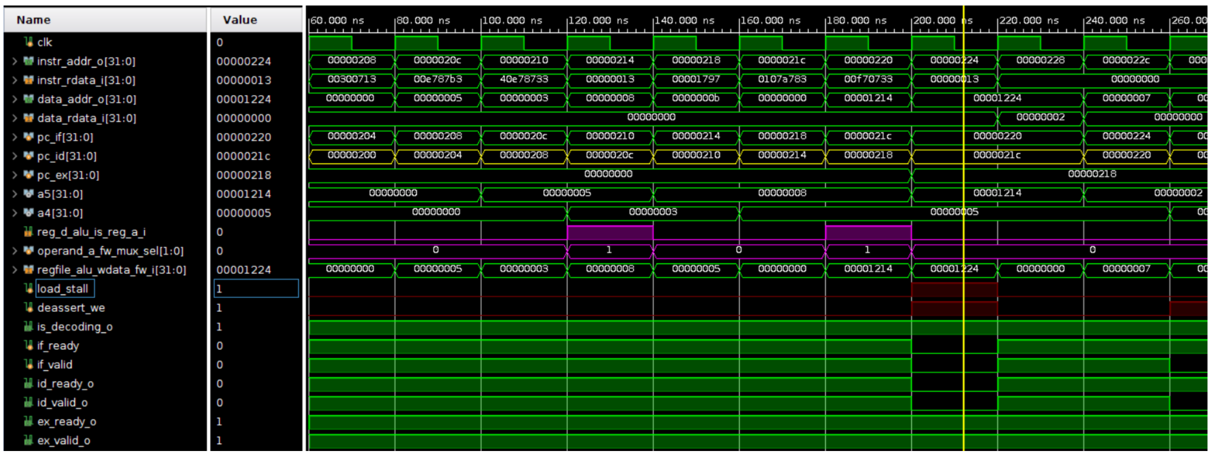
```
OUTPUT_ARCH ("riscv")
ENTRY(_start)
SECTIONS
{
.text :
{
. = 0x100;
*(.vector_user)
. = 0x11c;
*(.vector_timer_int)
. = 0x1C0;
*(.vector_machine)
. = 0x200;
*(.start)
*(.text)
}
.rodata :
{
. = 0x1000;
*(.rodata)
}
.data :
{
. = 0x400000;
*(.data)
}
}
```

3 Test Cases

3.1 Basic ALU operation with forwarding and load-use data hazard

Listing 2: Forwarding and load-use hazard

```
.text
.globl _start
_start:
        li a5, 5
        li a4, 3
        add a5, a5, a4
        sub a4, a5, a4 #a5 forwarded from EX
        nop
        lw a5, x
        add a4, a4, a5 #load-use data hazard
        nop

.section .rodata
x: .word 0x2

forward.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
 200:    00500793          li       a5,5
 204:    00300713          li       a4,3
 208:    00e787b3          add      a5,a5,a4
 20c:    40e78733          sub      a4,a5,a4
 210:    00000013          nop
 214:    00001797          auipc    a5,0x1
 218:    0107a783          lw       a5,16(a5) # 1224 <x>
 21c:    00f70733          add      a4,a4,a5
 220:    00000013          nop
```



Name	Value
clk	0
instr_addr_o[31:0]	00000224
instr_rdata_i[31:0]	00000013
data_addr_o[31:0]	00001224
data_rdata_i[31:0]	00000000
pc_if[31:0]	00000220
pc_id[31:0]	0000021c
pc_ex[31:0]	00000218
a5[31:0]	00001214
a4[31:0]	00000005
reg_d_alu_is_reg_a_i	0
operand_a_fw_mux_sel[1:0]	0
regfile_alu_wdata_fw_i[31:0]	00001224
load_stall	1
deassert_we	1
is_decoding_o	1
if_ready	0
if_valid	0
id_ready_o	0
id_valid_o	0
ex_ready_o	1
ex_valid_o	1

Figure 1: Forward

3.2 Flush on Branch

Listing 3: Branch

```
.text
.globl _start
_start:
        li a5, 0
        beq x0, a5, success #branch taken
        addi a4, x0, 3
        addi a4, x0, 5
        addi a4, x0, 7
success:
        addi a5, x0, 15
        beq x0, a5, success #branch not taken
        nop

flushBranch.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
 200:   00000793              li      a5,0
 204:   00f00863              beq     zero,a5,214 <success>
 208:   00300713              li      a4,3
 20c:   00500713              li      a4,5
 210:   00700713              li      a4,7
00000214 <success>:
 214:   00f00793              li      a5,15
 218:   fef00ee3              beq     zero,a5,214 <success>
 21c:   00000013              nop
```

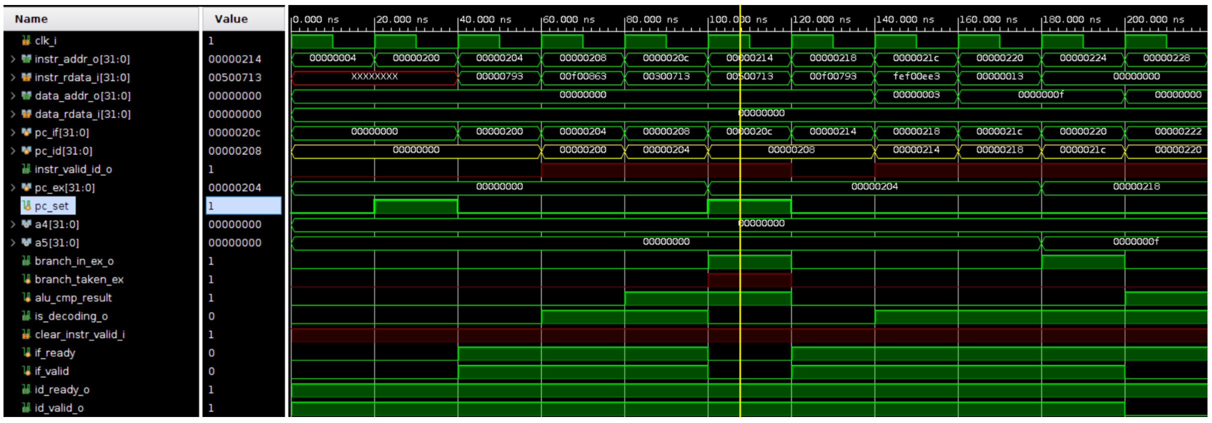


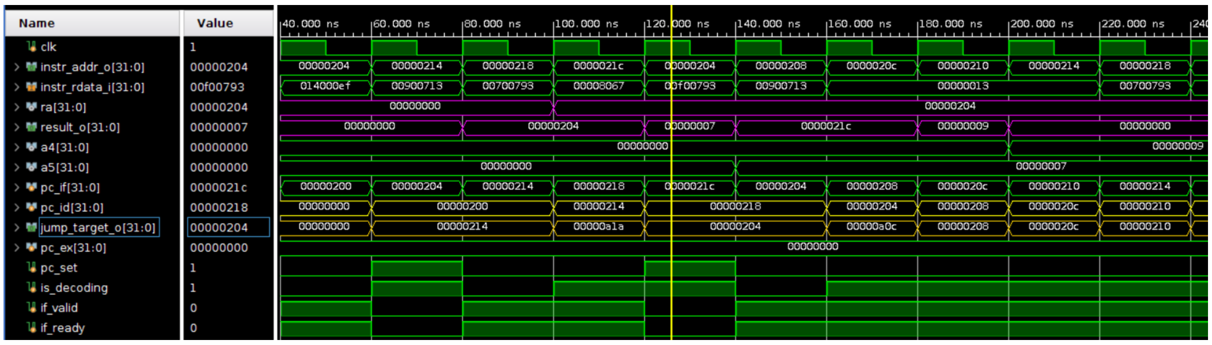
Figure 2: Flush on branch

3.3 Function Call

[MM: Maybe try one with parameter passing]

Listing 4: Function Call, jalr

```
.text
.globl _start
_start:
        jal ra, fun
        li a4, 9
        nop
        nop
        nop
fun:
        addi a5, x0, 7
        jalr x0, ra #return
        addi a5, x0, 15
        nop

jalr.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
 200:   014000ef                jal     ra,214 <fun>
 204:   00900713                li      a4,9
 208:   00000013                nop
 20c:   00000013                nop
 210:   00000013                nop

00000214 <fun>:
 214:   00700793                li      a5,7
 218:   00008067                ret
 21c:   00f00793                li      a5,15
 220:   00000013                nop
```



Name	Value
clk	1
instr_addr_o[31:0]	00000204
instr_rdata_i[31:0]	00f00793
ra[31:0]	00000204
result_o[31:0]	00000007
a4[31:0]	00000000
a5[31:0]	00000000
pc_if[31:0]	0000021c
pc_id[31:0]	00000218
jump_target_o[31:0]	00000204
pc_ex[31:0]	00000000
pc_set	1
is_decoding	1
if_valid	0
if_ready	0

Figure 3: Function Call

3.4 Instruction Fetch

Prefetch Buffer accesses the Instruction Memory. Address is word aligned. The signals (req, gnt, rvalid) are shown in the waveform.

3.5 LSU

LSU accesses the Data Memory. words (32-bit), half words (16-bit), byte (8-bit) are supported. Explain the *data_be* signal which specifies which bytes are read/written.
Misaligned memory accesses are also handled by the LSU within two bus transactions.

3.5.1 Loads

Load Byte:

Listing 5: Load Byte

```
.text
.globl _start
_start:
        lw a4, x
        lw a5, y
        add a3, a4, a5
        lb a4, z #a4 stores 0x23
        nop

.section .rodata
        x: .word 0x2
        y: .word 0x5
        z: .word 0x123

load.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
200:    00001717                auipc   a4,0x1
204:    02072703                lw      a4,32(a4) # 1220 <x>
208:    00001797                auipc   a5,0x1
20c:    01c7a783                lw      a5,28(a5) # 1224 <y>
210:    00f706b3                add     a3,a4,a5
214:    00001717                auipc   a4,0x1
218:    01470703                lb      a4,20(a4) # 1228 <z>
21c:    00000013                nop
```

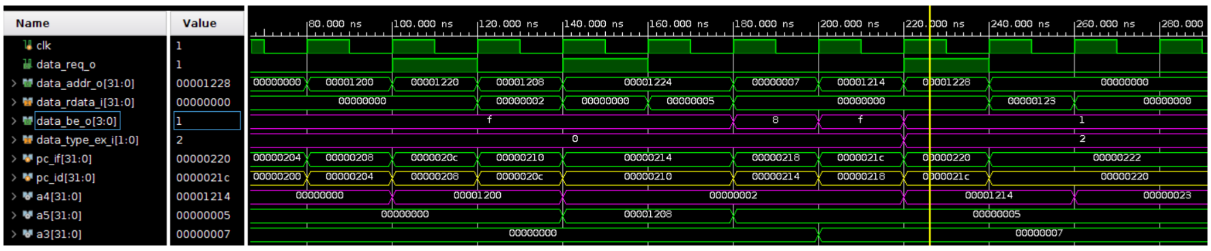


Figure 4: Load Byte

3.5.2 Store and Load

Listing 6: Loads after Stores

```
.text
.globl _start
_start:
        li a5, 2
        li a4, 3
        la a3, x
        sw a5, 0(a3)
        sw a4, 4(a3)
        lw a5, 4(a3)
        lw a4, 0(a3)
        nop
.data
        x: .word

stores.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
        200:    00200793            li      a5,2
        204:    00300713            li      a4,3
        208:    00401697            auipc   a3,0x401
        20c:    01c68693            addi    a3,a3,28 # 401224 <x>
        210:    00f6a023            sw      a5,0(a3)
        214:    00e6a223            sw      a4,4(a3)
        218:    0046a783            lw      a5,4(a3)
        21c:    0006a703            lw      a4,0(a3)
        220:    00000013            nop
```

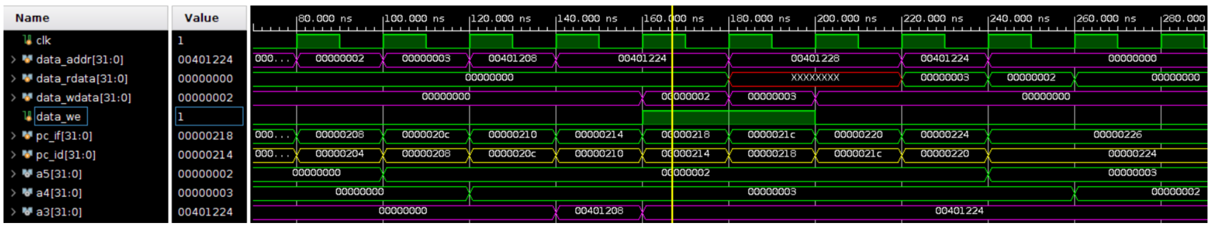


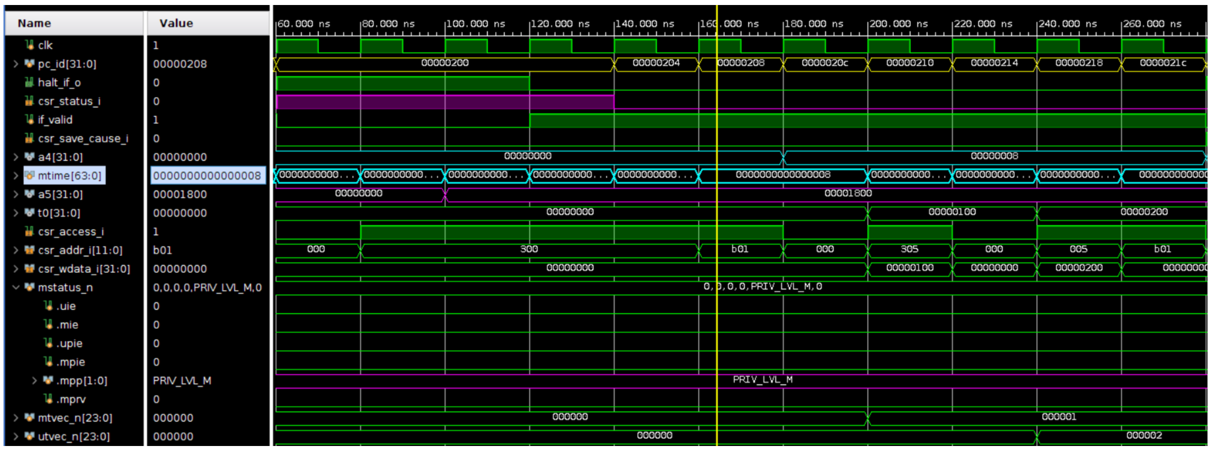
Figure 5: Loads after Stores

3.6 Control and Status Registers

[MM: Retry, utvec is not available in my modified core, but it is here just to show the 256-byte alignment for the exception vector]

Listing 7: CSR RW

```
.text
.globl _start
_start:
         csrr a5, mstatus  #0x300
         csrr a4, 0xb01  #mtime
         li t0, 0x100
         csrw mtvec, t0
         li t0, 0x200
         csrw 0x005, t0 #utvec
         csrr a4, 0xb01  #mtime
         nop

csrRW.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
200:     300027f3               csrr    a5,mstatus
204:     b0102773               csrr    a4,0xb01
208:     10000293               li      t0,256
20c:     30529073               csrw    mtvec,t0
210:     20000293               li      t0,512
214:     00529073               csrw    0x5,t0
218:     b0102773               csrr    a4,0xb01
21c:     00000013               nop
```



Name	Value
clk	1
pc_id[31:0]	00000208
halt_if_o	0
csr_status_i	0
if_valid	1
csr_save_cause_i	0
a4[31:0]	00000000
mtime[63:0]	0000000000000008
a5[31:0]	00001800
t0[31:0]	00000000
csr_access_i	1
csr_addr_i[11:0]	b01
csr_wdata_i[31:0]	00000000
mstatus_n	0,0,0,0,PRIV_LVL_M,0
.uie	0
.mie	0
.upie	0
.mpie	0
.mpp[1:0]	PRIV_LVL_M
.mprv	0
mtvec_n[23:0]	000000
utvec_n[23:0]	000000

Figure 6: csrRW

3.7 MRET and ECALL

mstatus Register.

With **mret** the core jumps to the address previously stored in **mepc**, and changes the privilege level to user, and enbles the interrupts. [MM: Enabling the interrupts is my modification in the cs_registers.sv]

From te Privilege Spec: 'The MRET instruction is used to return from traps in M-mode. When executing an MRET instruction, supposing MPP holds the value U, MIE is set to MPIE; the privilege mode is changed to U; MPIE is set to 1; and MPP is set to U (or M if user-mode is not supported).'

Listing 8: mret

```
csr_restore_mret_i: begin //MRET
        mstatus_n.mie   = mstatus_q.mpie; // on init mpie holds 1
        priv_lvl_n      = PRIV_LVL_U;     // and mpp holds U-mode
        mstatus_n.mpie  = 1'b1;
        mstatus_n.mpp   = PRIV_LVL_U;
end
```

After **ecall** the core jumps to the address stored in **mtvec**. At the same time the address of the **ecall** instruction is saved bu the core in **mepc**.

From te Privilege Spec: 'MPIE holds the value of the interrupt-enable bit active prior to the trap, and MPP holds the previous privilege mode. The MPP fields can only hold privilege modes up to M, so MPP is two bits wide, SPP is one bit wide, and UPP is implicitly zero. When a trap is taken from privilege mode U into privilege mode M, MPIE is set to the value of MIE; MIE is set to 0; and MPP is set to U.'

Listing 9: ecall

```
priv_lvl_n      = PRIV_LVL_M;
mstatus_n.mpie  = mstatus_q.mie;
mstatus_n.mie   = 1'b0;
mstatus_n.mpp   = PRIV_LVL_U;
mepc_n          = exception_pc;
mcause_n        = csr_cause_i;
```

Listing 10: mret

```
.section .vector_user #jump here after exception
        j       vector_u
.text


.globl _start
_start:
        li t0, 0x100
        csrw  mtvec, t0
        la t0, user_mode #where to go after mret
        csrw mepc, t0

        mret #jump to the address in mepc
        li a5, 7
        nop

.globl user_mode
user_mode:
        li a5, 5
        ecall  #jump to the address in mtvec
        nop

.globl vector_u
vector_u:
        li a5, 7
        mret
        nop


mret.elf:      file format elf32-littleriscv


Disassembly of section .text:
00000000 <_start-0x200>:...
100:    12c0006f                j       22c <vector_u>
...
00000200 <_start>:
200:    10000293                li      t0,256
204:    30529073                csrw    mtvec,t0
208:    00000297                auipc   t0,0x0
20c:    01828293                addi    t0,t0,24 # 220 <user_mode>
210:    34129073                csrw    mepc,t0
214:    30200073                mret
218:    00700793                li      a5,7
21c:    00000013                nop

00000220 <user_mode>:
220:    00500793                li      a5,5
224:    00000073                ecall
228:    00000013                nop

0000022c <vector_u>:
22c:    00700793                li      a5,7
230:    30200073                mret
234:    00000013                nop
```

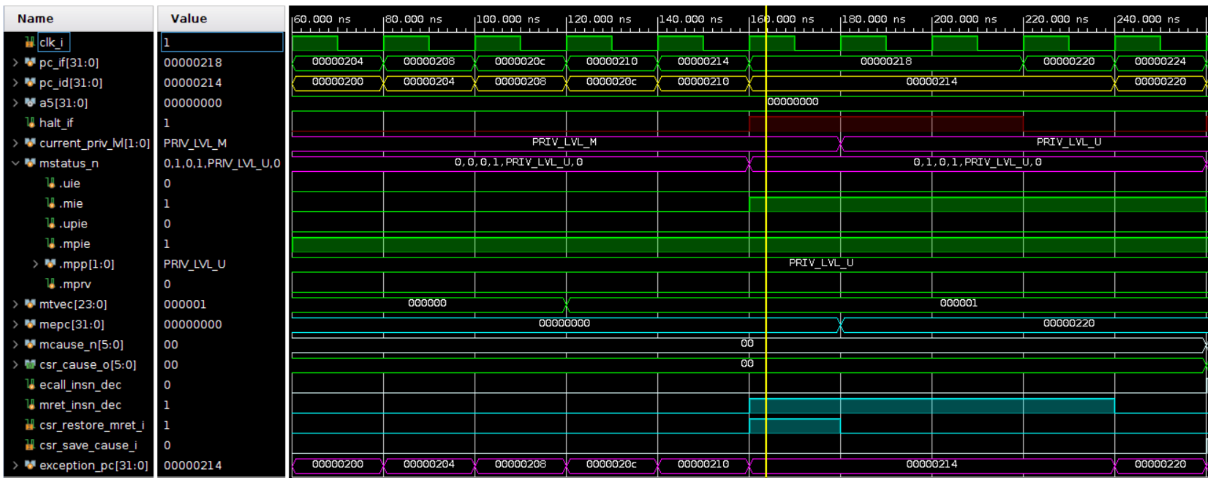


Figure 7: mret

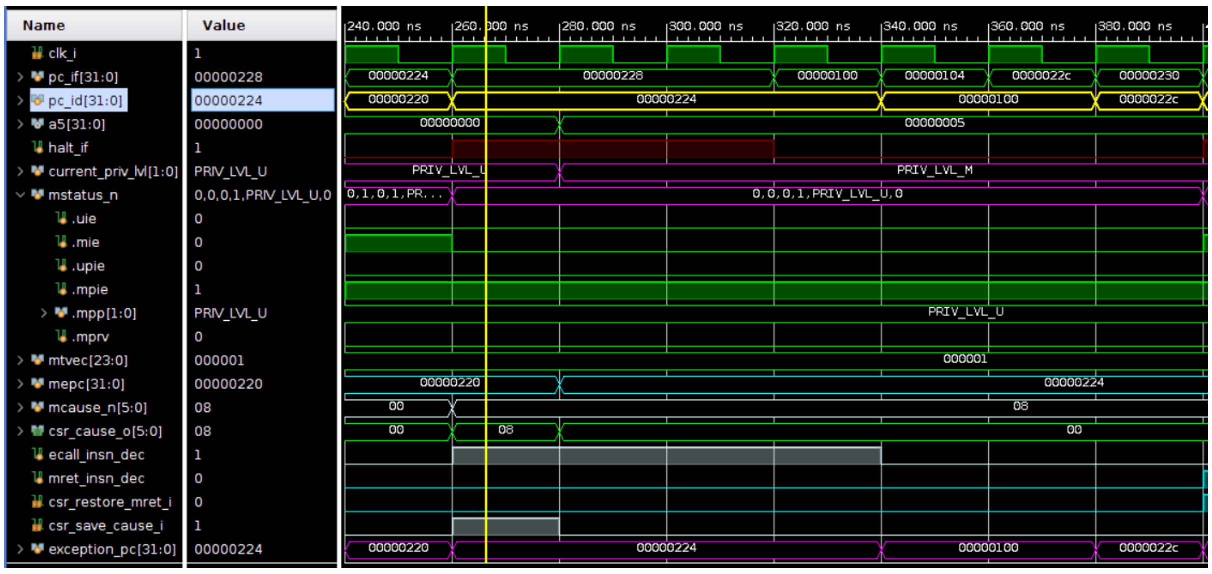


Figure 8: ecall, Note mepc is changed

3.8 Interrupts

mstatus[MIE] == 1 and mie == 1
highest ID = highest priority, MTI has lowest priority: *irg_i[31], irg_i[30], ..., irg_i[16], irg_i[11], irg_i[3], irg_i[7]*
3 Machine software interrupt
7 Machine timer interrupt
11 Machine external interrupt
0, 4, 8 User interrupts
1, 5, 9 Supervisor interrupts
15-12, 10, 6, 2 Reserved for future standard use
>=16 Reserved for platform use

[MM: Rewrite this part, better!]
We first enable the timer interrupt in **mie** register and set a value in **mtimecmp** register. When the value in **mtimecmp** register becomes equal to the value in **mtime** register,

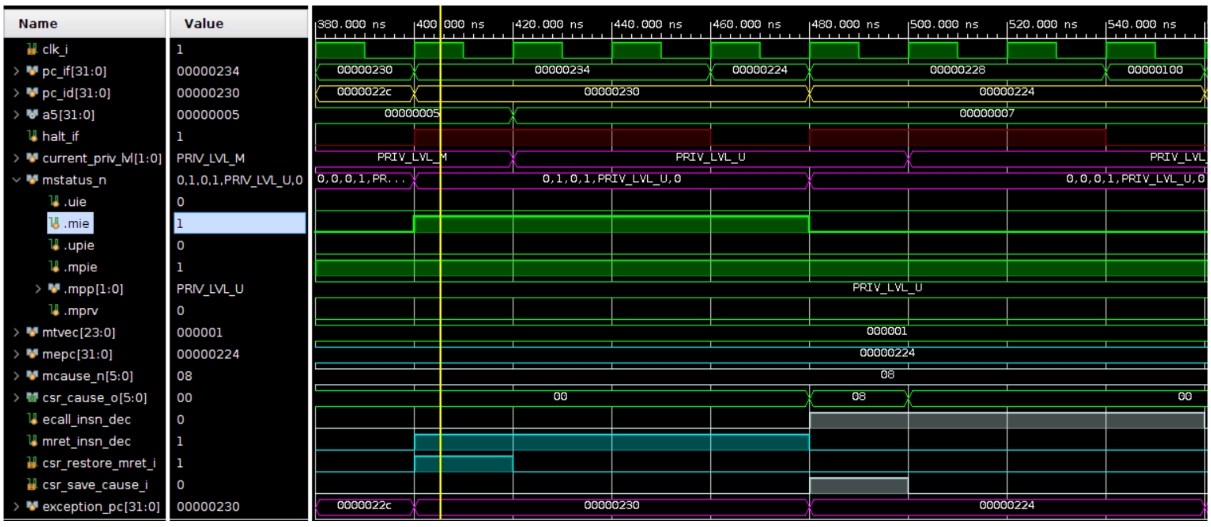


Figure 9: mret, Note it jumps to the changed mepc

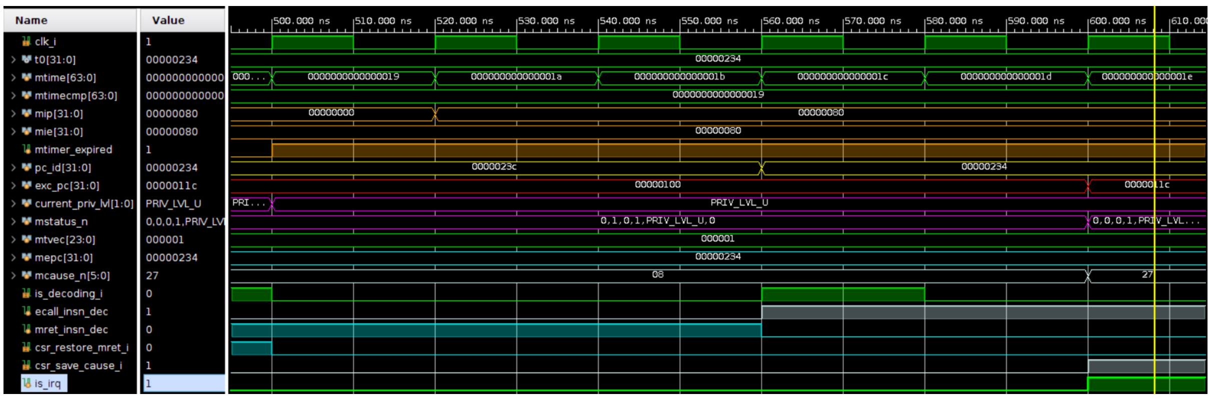
mtimer_expired signal becomes active, but the timer interrupt is only generated when the core is in U-mode (when mie bit in **mstatus** register is '1').

The core jumps to the timer exception address, i.e., 0x11c (see linker script) and from there as indicated in the assembly it jumps to the timer ISR (*timer_isr*). We clear the timer bit in the **mip** by writing to the **mtimecmp** register. We set it again, 15 cycles more than the current timer value. The whole sequence gets repeated. [MM: Note that we do not write to the mtimecmpH, so the code will work with 32-bit timer value]

Listing 11: Timer

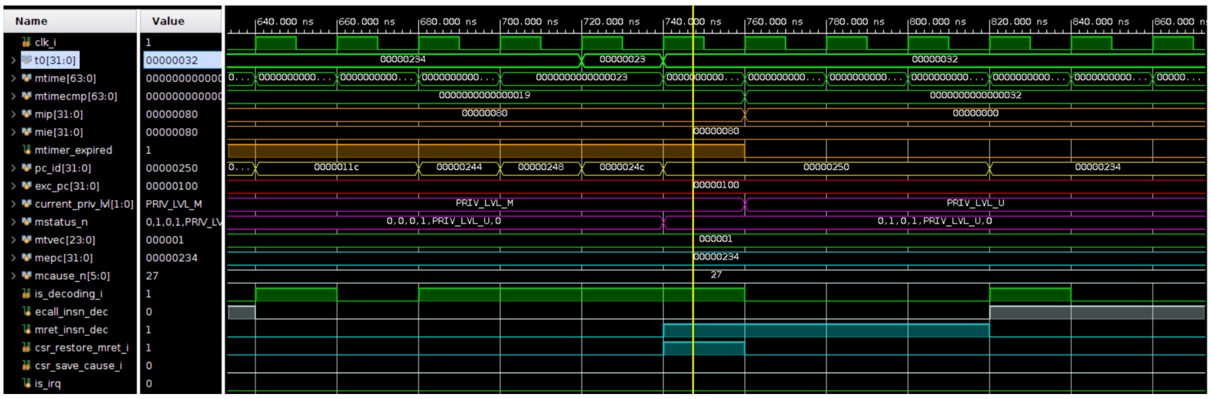
```
.section .vector_user #jump here after exception
        j       vector_u

.section .vector_timer_int
        j       timer_isr

.text
.globl _start
_start:
        li t0, 0x100
        csrw  mtvec, t0
        la t0, user_mode #where to go after mret
        csrw mepc, t0

        #timer-related instructions:
        li a4, 25
        li a5, 0
        csrw 0x322, a5
        csrw 0x321, a4 #write to mtimecmp
        li a4, 0x80
        csrs mie, a4 #enable timer interrupt

        mret #jump to the address in mepc
        nop

.globl user_mode
user_mode:
        ecall  #jump to the address in mtvec
        nop

.globl vector_u
vector_u:
        mret
        nop

.globl timer_isr
timer_isr:
        csrr t0, 0xb01  #mtime
        add t0, t0, 15
        csrw 0x321, t0 #add several cycles
        mret

mret.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
100:    13c0006f                j       23c <vector_u>
...
11c:    1280006f                j       244 <timer_isr>
...

00000200 <_start>:
200:    10000293                li      t0,256
204:    30529073                csrw    mtvec,t0
208:    00000297                auipc   t0,0x0
20c:    02c28293                addi    t0,t0,44 # 234 <user_mode>
210:    34129073                csrw    mepc,t0
```

```
 214:      01900713                li      a4,25
 218:      00000793                li      a5,0
 21c:      32279073                csrw    0x322,a5
 220:      32171073                csrw    0x321,a4
 224:      08000713                li      a4,128
 228:      30472073                csrs    0x304,a4
 22c:      30200073                mret
 230:      00000013                nop

00000234 <user_mode>:
 234:      00000073                ecall
 238:      00000013                nop

0000023c <vector_u>:
 23c:      30200073                mret
 240:      00000013                nop

00000244 <timer_isr>:
 244:      b01022f3                csrr    t0,0xb01
 248:      00f28293                addi    t0,t0,15
 24c:      32129073                csrw    0x321,t0
 250:      30200073                mret
```



Name	Value
clk_i	1
t0[31:0]	00000234
mtime[63:0]	000000000000
mtimecmp[63:0]	000000000000
mip[31:0]	00000080
mie[31:0]	00000080
mtimer_expired	1
pc_id[31:0]	00000234
exc_pc[31:0]	0000011c
current_priv_lvl[1:0]	PRIV_LVL_U
mstatus_n	0,0,0,1,PRIV_LV
mtvec[23:0]	000001
mepc[31:0]	00000234
mcause_n[5:0]	27
is_decoding_i	0
ecall_insn_dec	1
mret_insn_dec	0
csr_restore_mret_i	0
csr_save_cause_i	1
is_irq	1

Figure 10: Timer Expired an timer interrupt



Name	Value
clk_i	1
t0[31:0]	00000032
mtime[63:0]	000000000000
mtimecmp[63:0]	000000000000
mip[31:0]	00000080
mie[31:0]	00000080
mtimer_expired	1
pc_id[31:0]	00000250
exc_pc[31:0]	00000100
current_priv_lvl[1:0]	PRIV_LVL_M
mstatus_n	0,1,0,1,PRIV_LV
mtvec[23:0]	000001
mepc[31:0]	00000234
mcause_n[5:0]	27
is_decoding_i	1
ecall_insn_dec	0
mret_insn_dec	1
csr_restore_mret_i	1
csr_save_cause_i	0
is_irq	0

Figure 11: TimerISR and reprogram

3.9 GPIO

Listing 12: GPIO Led on

```
.text
.globl _start
_start:
        li a5, 0x1

        lui a6, %hi(8388864) #GPIO_OUT = 0x00800100 = 8,388,864 dec
        addi a6, a6, %lo(8388864)

        sw a5, 0(a6)
        li a5, 0
        loop:
        beq x0, a5, loop



gpio.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
200:    00100793            li      a5,1
204:    00800837            lui     a6,0x800
208:    10080813            addi    a6,a6,256 # 800100 <loop+0x7ffeec>
20c:    00f82023            sw      a5,0(a6)
210:    00000793            li      a5,0
00000214 <loop>:
214:    00f00063            beq     zero,a5,214 <loop>
```

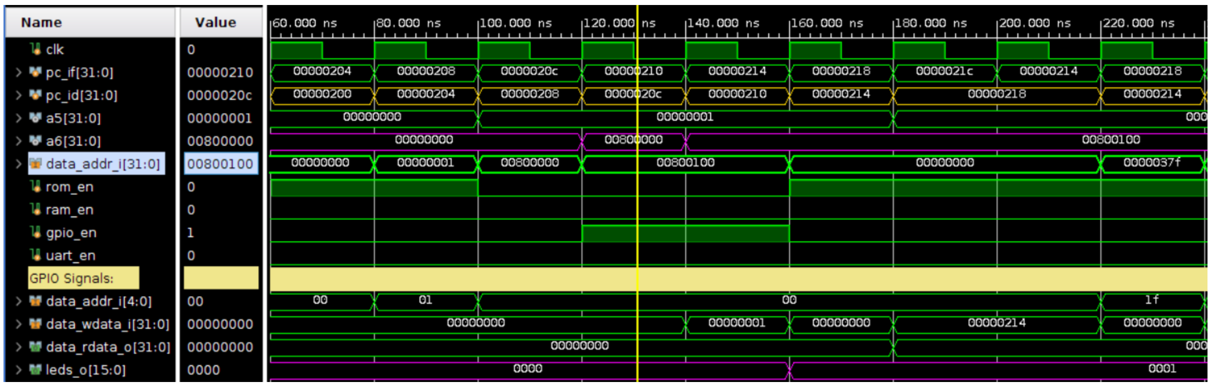


Figure 12: GPIO led0 on

Listing 13: GPIO Led on after switch on

```
.text
.globl _start
_start:

        lui a6, %hi(8388864) #GPIO_OUT = 0x00800100 = 8,388,864 dec
        addi a6, a6, %lo(8388864) s

        lui a4, %hi(8388868) #GPIO_IN = 0x00800104 = 8,388,868 dec
        addi a4, a4, %lo(8388868)

        li t0, 0
        loop:
        lw a5, 0(a4) #get the value from the switches
        sw a5, 0(a6) #turn the corresponding leds on
        beq x0, t0, loop


gpio.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
200:    00800837                lui     a6,0x800
204:    10080813                addi    a6,a6,256 # 800100 <loop+0x7ffeec>
208:    00800737                lui     a4,0x800
20c:    10470713                addi    a4,a4,260 # 800104 <loop+0x7ffef0>
210:    00000293                li      t0,0

00000214 <loop>:
214:    00072783                lw      a5,0(a4)
218:    00f82023                sw      a5,0(a6)
21c:    fe500ce3                beq     zero,t0,214 <loop>
```

3.10 UART

[MM: Read the vscale ppt, the transmitted bit figure might be wrong]

Listing 14: GPIO Led on after switch on

```
.text
.globl _start
_start:
        #UART_CTRL =  BAUD_DIV << 16;
        lui     a5,0x800
        addi a5,a5,512 # 800200 -> UART_CTRL

        #0x1b10  => baud rate
        lui     a4,0x1b10
        sw      a4,0(a5)


        #while(UART_STATUS & 0x0200);
        #0x200 = 001000000000
        #while (tx_full);


        loop: lui a5,0x800
        addi a5,a5,516 # 800204 ->

        lw      a5,0(a5)
        andi a5,a5,512 #512 = 0x200
        bnez a5,loop #if UART is full

        # char c = 'M';
        # UART_DATA = (unsigned char)c;

        lui     a5,0x800
        addi a5,a5,520 # 800208 -> UART_DATA

        li      a4,77 #77 = 'M' = 0x4d
        sw      a4,0(a5)

uart.elf:     file format elf32-littleriscv
Disassembly of section .text:
00000000 <_start-0x200>:
...
00000200 <_start>:
200:    008007b7               lui     a5,0x800
204:    20078793               addi    a5,a5,512 # 800200 <loop+0x7ffff0>
208:    01b10737               lui     a4,0x1b10
20c:    00e7a023               sw      a4,0(a5)

00000210 <loop>:
210:    008007b7               lui     a5,0x800
214:    20478793               addi    a5,a5,516 # 800204 <loop+0x7ffff4>
218:    0007a783               lw      a5,0(a5)
21c:    2007f793               andi    a5,a5,512
220:    fe0798e3               bnez    a5,210 <loop>
224:    008007b7               lui     a5,0x800
228:    20878793               addi    a5,a5,520 # 800208 <loop+0x7ffff8>
22c:    04d00713               li      a4,77
230:    00e7a023               sw      a4,0(a5)
```

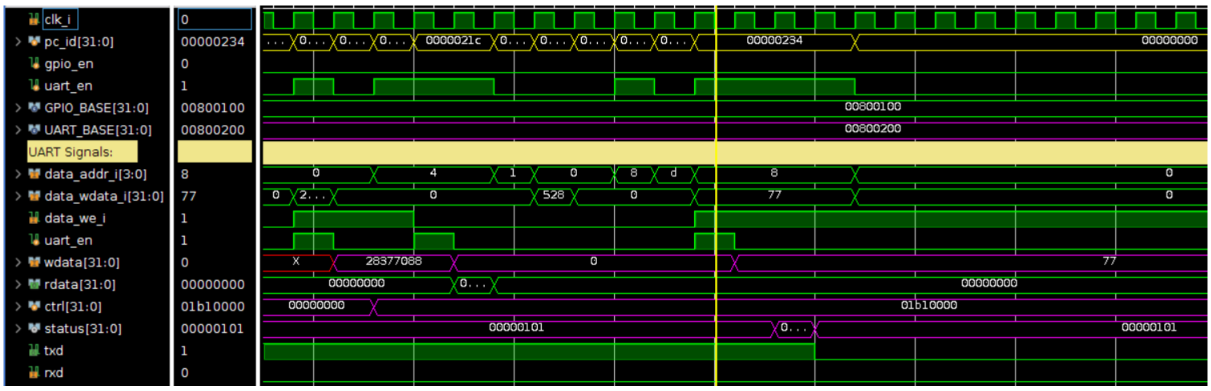


Figure 13: UART

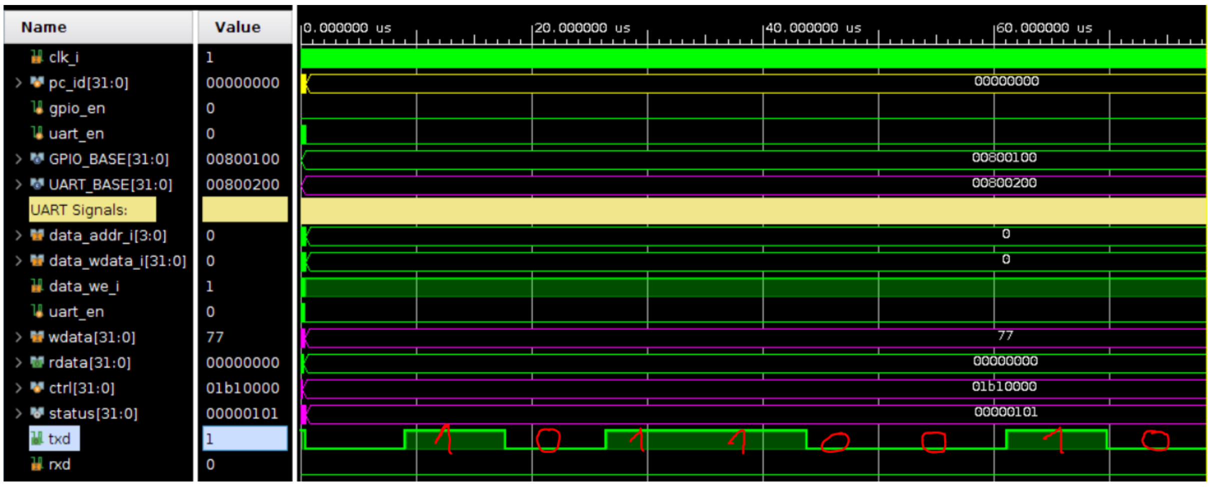
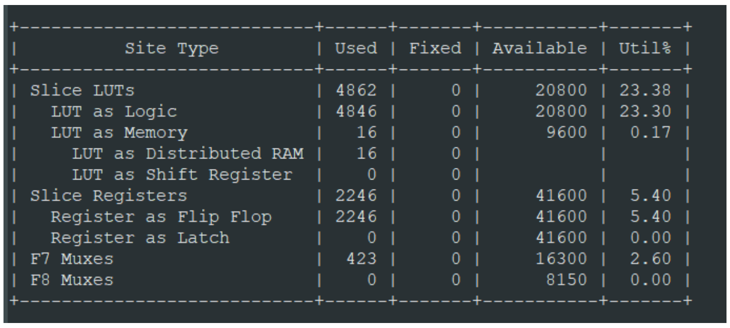


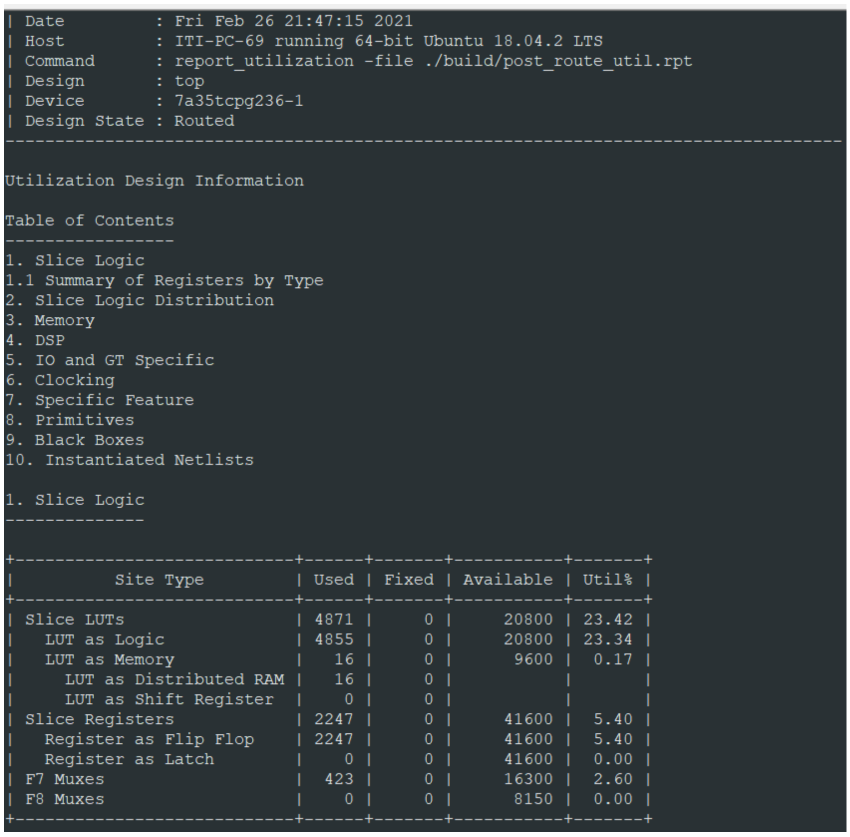
Figure 14: Char 'M' = 77d = 'b0100 1101

4 Evaluation



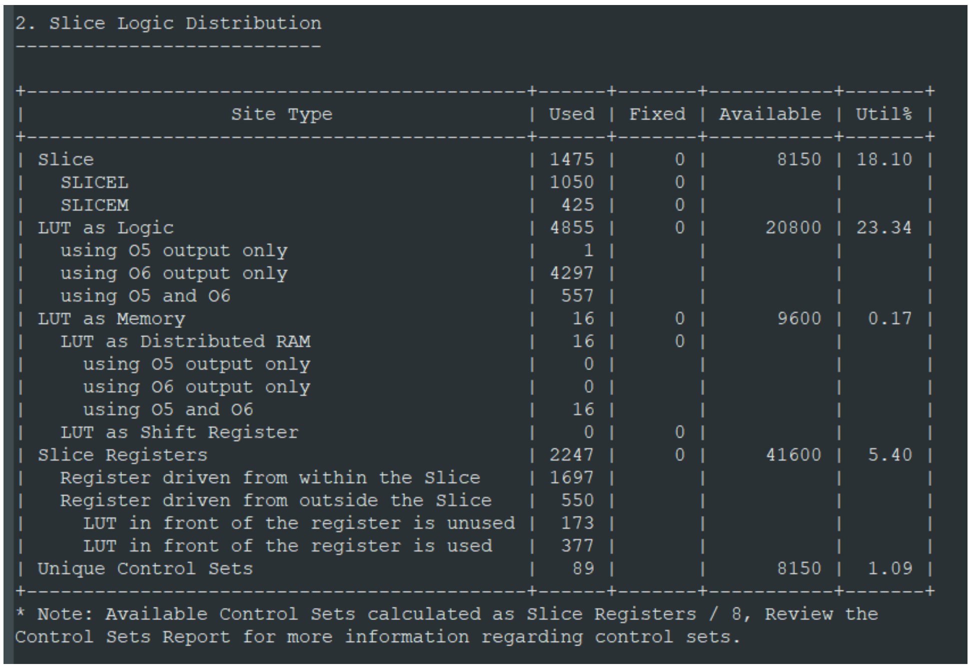
```
+----------------------------+------+-------+-----------+-------+
|         Site Type          | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 | 4862 |     0 |     20800 | 23.38 |
|   LUT as Logic             | 4846 |     0 |     20800 | 23.30 |
|   LUT as Memory            |   16 |     0 |      9600 |  0.17 |
|     LUT as Distributed RAM |   16 |     0 |           |       |
|     LUT as Shift Register  |    0 |     0 |           |       |
| Slice Registers            | 2246 |     0 |     41600 |  5.40 |
|   Register as Flip Flop    | 2246 |     0 |     41600 |  5.40 |
|   Register as Latch        |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                   |  423 |     0 |     16300 |  2.60 |
| F8 Muxes                   |    0 |     0 |      8150 |  0.00 |
+----------------------------+------+-------+-----------+-------+
```

Figure 15: postroute



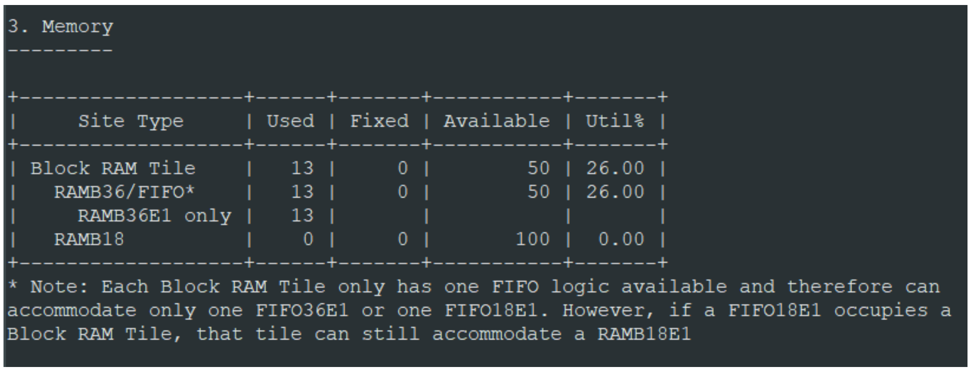
```
| Date         : Fri Feb 26 21:47:15 2021
| Host         : ITI-PC-69 running 64-bit Ubuntu 18.04.2 LTS
| Command      : report_utilization -file ./build/post_route_util.rpt
| Design       : top
| Device       : 7a35tcpg236-1
| Design State : Routed
------------------------------------------------------------------------------------

Utilization Design Information

Table of Contents
-----------------
1. Slice Logic
1.1 Summary of Registers by Type
2. Slice Logic Distribution
3. Memory
4. DSP
5. IO and GT Specific
6. Clocking
7. Specific Feature
8. Primitives
9. Black Boxes
10. Instantiated Netlists

1. Slice Logic
--------------

+----------------------------+------+-------+-----------+-------+
|         Site Type          | Used | Fixed | Available | Util% |
+----------------------------+------+-------+-----------+-------+
| Slice LUTs                 | 4871 |     0 |     20800 | 23.42 |
|   LUT as Logic             | 4855 |     0 |     20800 | 23.34 |
|   LUT as Memory            |   16 |     0 |      9600 |  0.17 |
|     LUT as Distributed RAM |   16 |     0 |           |       |
|     LUT as Shift Register  |    0 |     0 |           |       |
| Slice Registers            | 2247 |     0 |     41600 |  5.40 |
|   Register as Flip Flop    | 2247 |     0 |     41600 |  5.40 |
|   Register as Latch        |    0 |     0 |     41600 |  0.00 |
| F7 Muxes                   |  423 |     0 |     16300 |  2.60 |
| F8 Muxes                   |    0 |     0 |      8150 |  0.00 |
+----------------------------+------+-------+-----------+-------+
```

Figure 16: postroute



```
2. Slice Logic Distribution
---------------------------

+-------------------------------------------+------+-------+-----------+-------+
|                 Site Type                 | Used | Fixed | Available | Util% |
+-------------------------------------------+------+-------+-----------+-------+
| Slice                                     | 1475 |     0 |      8150 | 18.10 |
|   SLICEL                                  | 1050 |     0 |           |       |
|   SLICEM                                  |  425 |     0 |           |       |
| LUT as Logic                              | 4855 |     0 |     20800 | 23.34 |
|   using O5 output only                    |    1 |       |           |       |
|   using O6 output only                    | 4297 |       |           |       |
|   using O5 and O6                         |  557 |       |           |       |
| LUT as Memory                             |   16 |     0 |      9600 |  0.17 |
|   LUT as Distributed RAM                  |   16 |     0 |           |       |
|     using O5 output only                  |    0 |       |           |       |
|     using O6 output only                  |    0 |       |           |       |
|     using O5 and O6                       |   16 |       |           |       |
|   LUT as Shift Register                   |    0 |     0 |           |       |
| Slice Registers                           | 2247 |     0 |     41600 |  5.40 |
|   Register driven from within the Slice   | 1697 |       |           |       |
|   Register driven from outside the Slice  |  550 |       |           |       |
|     LUT in front of the register is unused |  173 |       |           |       |
|     LUT in front of the register is used  |  377 |       |           |       |
| Unique Control Sets                       |   89 |       |      8150 |  1.09 |
+-------------------------------------------+------+-------+-----------+-------+
* Note: Available Control Sets calculated as Slice Registers / 8, Review the
Control Sets Report for more information regarding control sets.
```

Figure 17: logicDistribution



```
3. Memory
---------

+-------------------+------+-------+-----------+-------+
|     Site Type     | Used | Fixed | Available | Util% |
+-------------------+------+-------+-----------+-------+
| Block RAM Tile    |   13 |     0 |        50 | 26.00 |
|   RAMB36/FIFO*    |   13 |     0 |        50 | 26.00 |
|     RAMB36E1 only |   13 |       |           |       |
|   RAMB18          |    0 |     0 |       100 |  0.00 |
+-------------------+------+-------+-----------+-------+
* Note: Each Block RAM Tile only has one FIFO logic available and therefore can
accommodate only one FIFO36E1 or one FIFO18E1. However, if a FIFO18E1 occupies a
Block RAM Tile, that tile can still accommodate a RAMB18E1
```

Figure 18: memoryDistribution