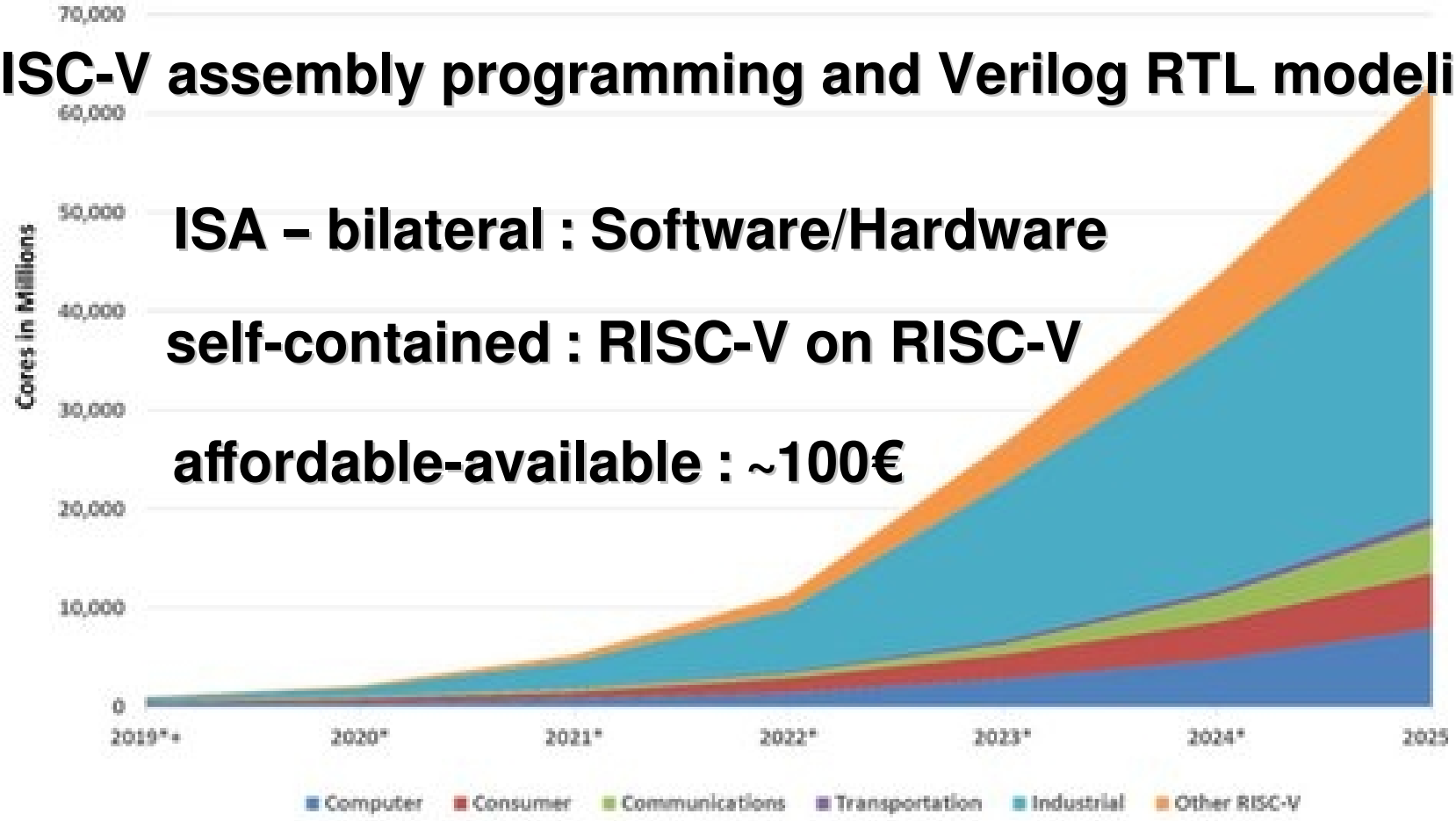
Programming and Modeling RISC-V ISA

RISC-V assembly programming and Verilog RTL modeling

ISA – bilateral : Software/Hardware

self-contained : RISC-V on RISC-V

affordable-available : ~100€



RISC-V cores

Source: Semico Research Corp.

RVLabs – RVPLabs & RVMLabs

RVPLabs - Software – ISA

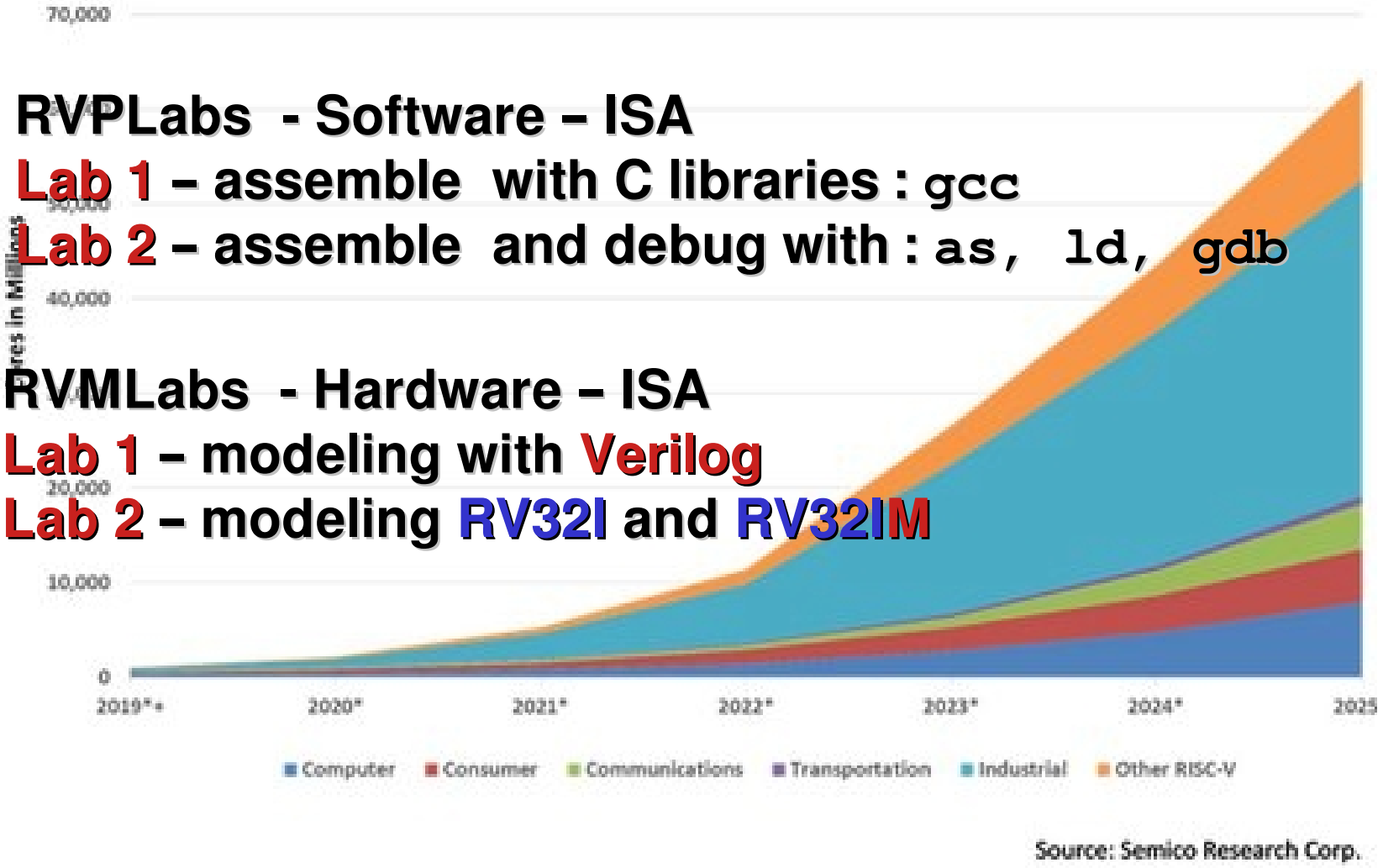
Lab 1 – assemble with C libraries : `gcc`

Lab 2 – assemble and debug with : `as, ld, gdb`

RVMLabs - Hardware – ISA

Lab 1 – modeling with Verilog

Lab 2 – modeling RV32I and RV32IM



RISC-V : Software ISA (assembly)

C programming + intrinsics (V)

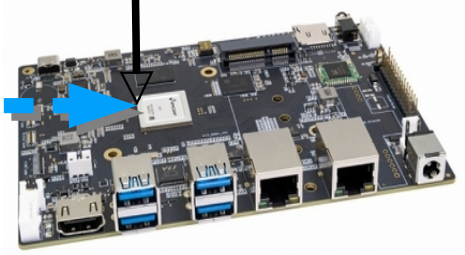
Assembly level programming

USER – binaries

RV64

OS

SpacemiT K1 – X60



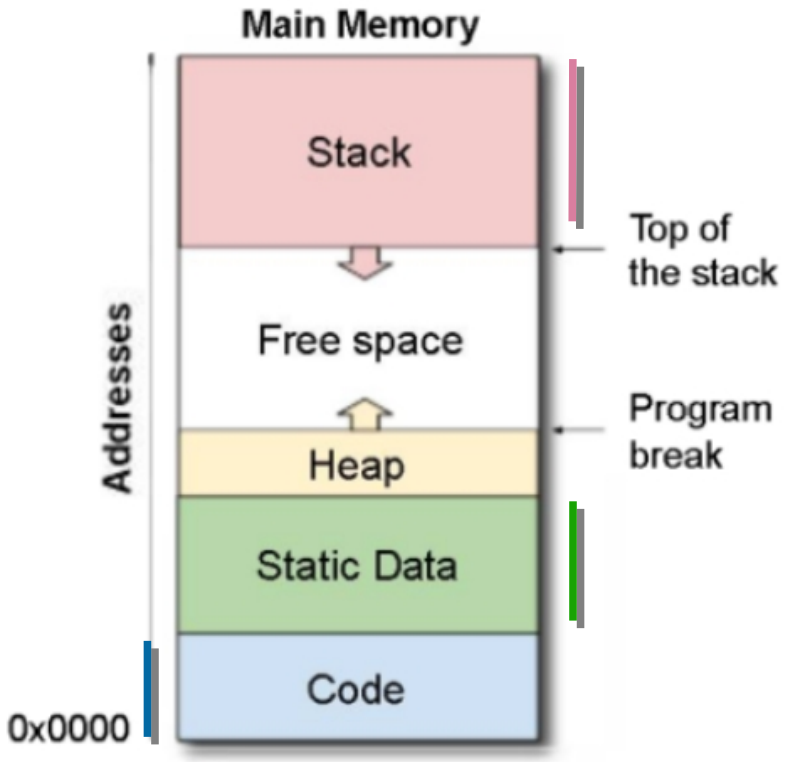
RISC-V : Hardware ISA (Verilog)

RTL – Verilog : busses,registers,ALUs,memories,decoders,..

Logic – Verilog: and,or,xor,nand,..

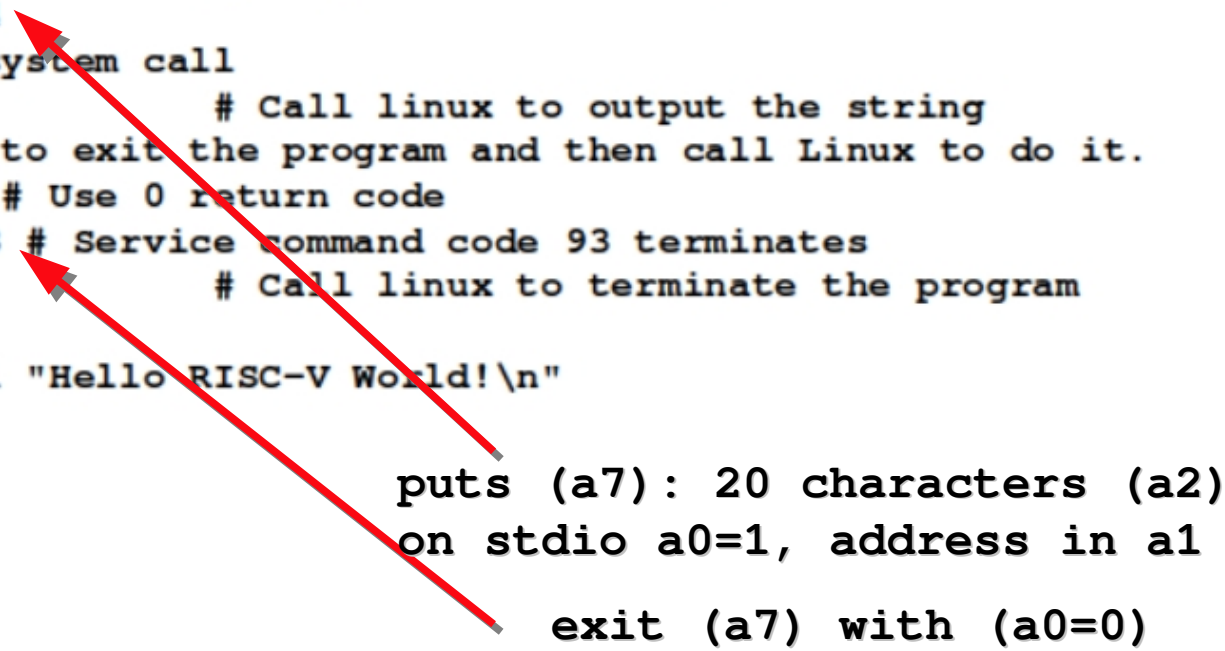
PLab1 : assemble with gcc

```
.LC0:
        .string "HelloWorld"
        .text
        .align  1
        .globl  main
        .type   main, @function
main:
.LFB6:
        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)
        lla     a0,.LC0
        call    puts@plt
        li      a0,0
        call    exit@plt
```



PLab2 : as, ld and gdb

```
.global main                        # Provide program starting address to linker
# Setup the parameters to print hello world and then call Linux to do it.
main:
        addi a0, x0, 1              # 1 =  stdout
        la a1, helloworld           # load address of helloworld
        addi a2, x0, 20             # length of our string
        addi a7, x0, 64
        # linux write system call
        ecall                       # Call linux to output the string
# Setup the parameters to exit the program and then call Linux to do it.
        addi a0, x0, 0 # Use 0 return code
        addi a7, x0, 93 # Service command code 93 terminates
        ecall                       # Call linux to terminate the program
.data
helloworld:      .ascii "Hello RISC-V World!\n"
```



PLabs : arithmetics

MultTen.s

```
.data
        .string "Result=%d\n"
.globl  main

main:
        li      a5,4
        mv      a4,a5
        slliw   a5,a5,2
        addw    a5,a5,a4
        slliw   a5,a5,1
        mv      a1,a5
        lla     a0,.data
        call    printf@plt  # link to physical address
        li      a0,0
        call    exit@plt
```

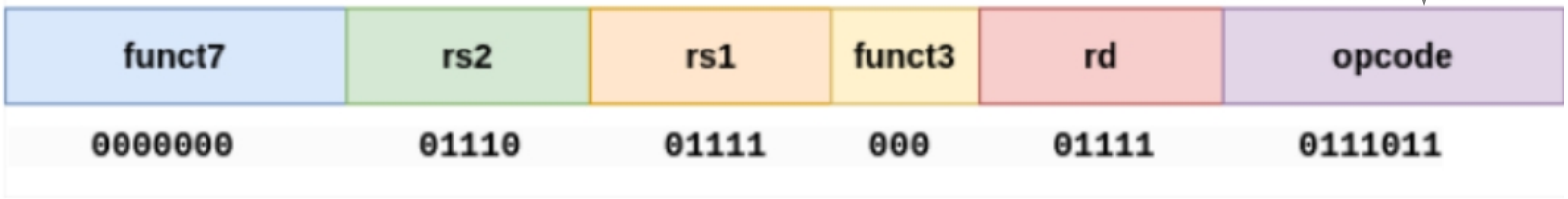
only add and shift

```
gcc MultTen.s -g -nostartfiles -o MultTen

riscv64-linux-gnu-objdump -d  MultTen
```

PLabs : arithmetics

```
00000000000006b8 <main>:          # virtual address - 24 bytes
 6b8:   00400793                  li      a5,4
 6bc:   00078713                  mv      a4,a5
 6c0:   0027979b                  sllw    a5,a5,0x2
 6c4:   00e787bb                  addw    a5,a5,a4
 6c8:   0017979b                  sllw    a5,a5,0x1
 6cc:   00078593                  mv      a1,a5
```

RV64I - R-type : 00e787bb addw a5,a5,a4



RV32I ISA - formats

ALUop: 0000000

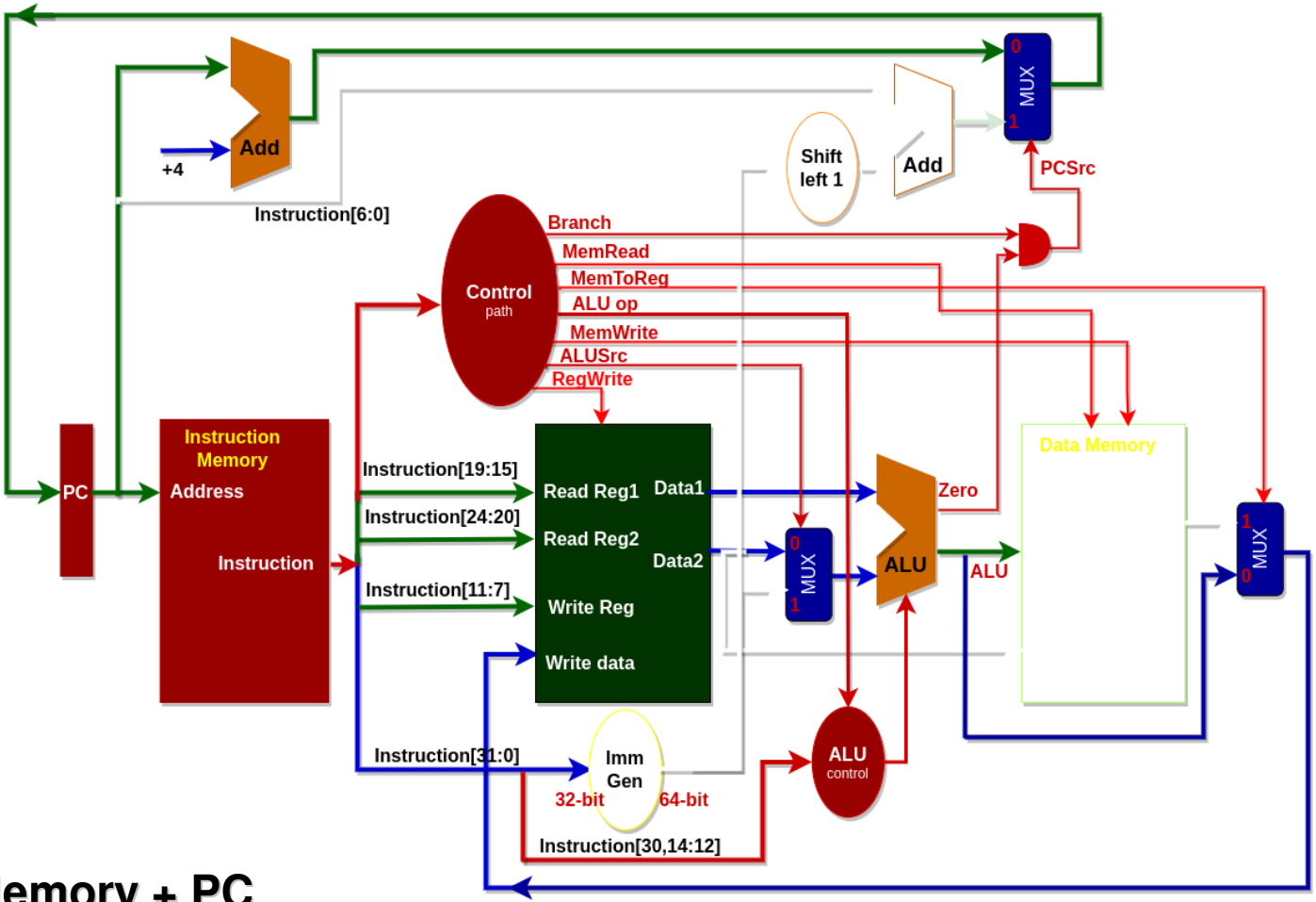
32-bit RISC-V Instruction Formats

Instruction Formats	31	30	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
R Register/register	funct7			rs2					rs1					funct3			rd					opcode						
I Immediate	imm[11:0]								rs1					funct3			rd					opcode						
U Upper Immediate	imm[31:12]																rd					opcode						
S Store	imm[11:5]			rs2					rs1					funct3			imm[4:0]					opcode						
B Branch	[12]	imm[10:5]		rs2					rs1					funct3			imm[4:1]			[11]		opcode						
J Jump	[20]	imm[10:1]						[11]	imm[19:12]								rd					opcode						

- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

```
addw a5,a5,a4
addw x15,x15,x14
```

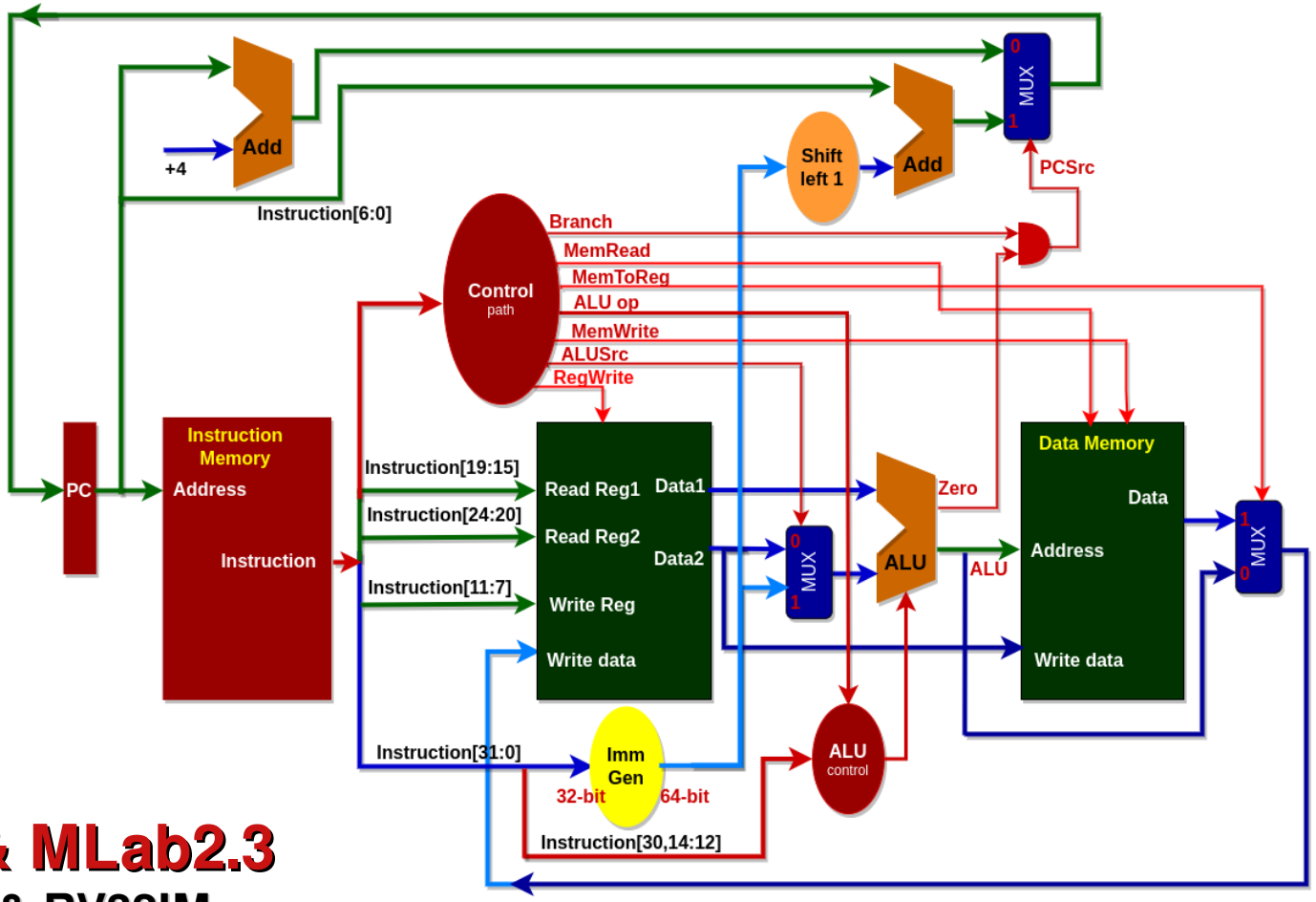
MLabs : Verilog (1) & Modeling (2-..)



MLab2.1

R-type only:
Instruction Memory + PC
Registers +ALU

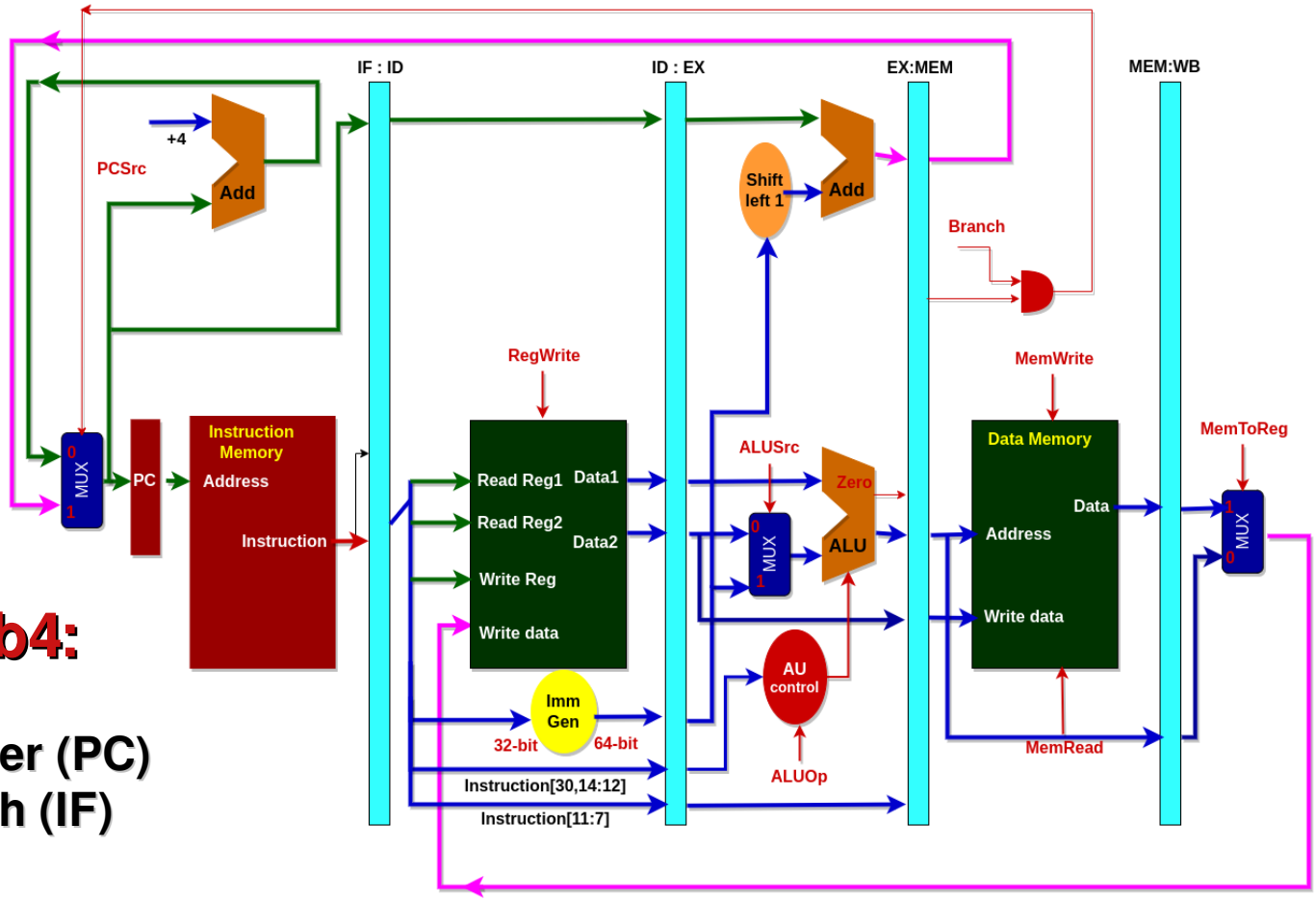
MLab 2.2 & 2.3 : RV32I & RV32IM



MLab2.2 & MLab2.3

RV32I & RV32IM

MLabs3,4 .. : RV32I pipeline



MLab3, MLab4:

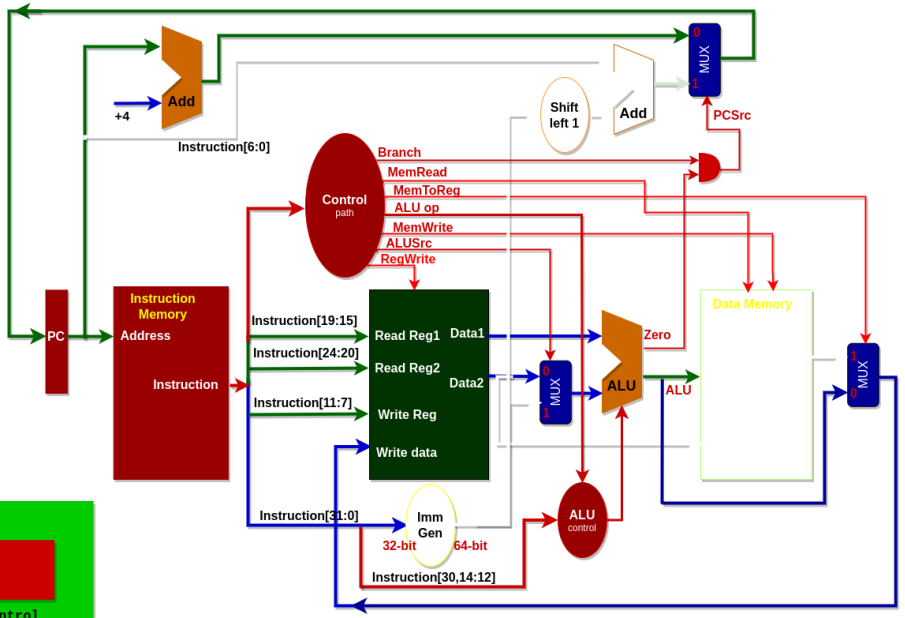
5-stage pipeline

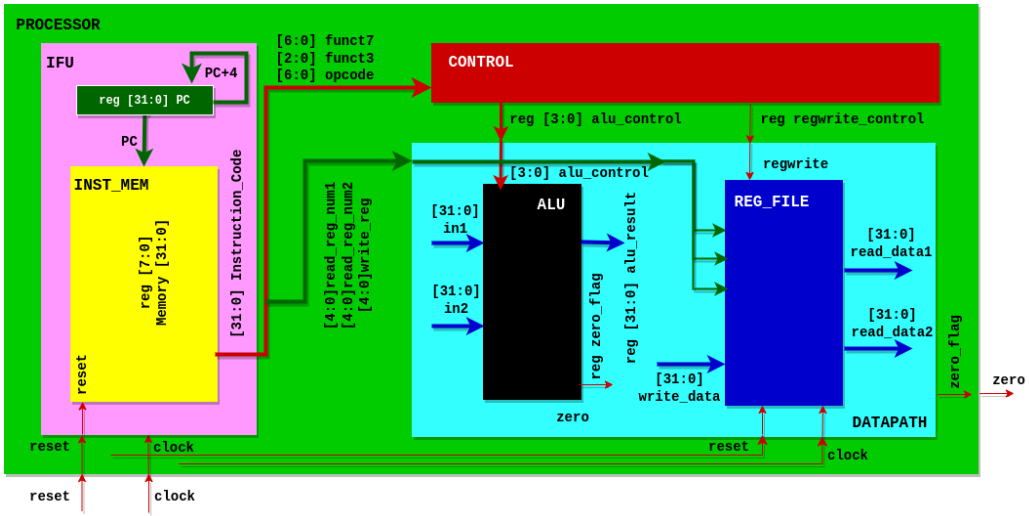
- Program Counter (PC)
- InstructionFetch (IF)
- DataRead (DR)
- Execute (EX)
- WriteBack (WB)

MLab2.1 : First design: R-type

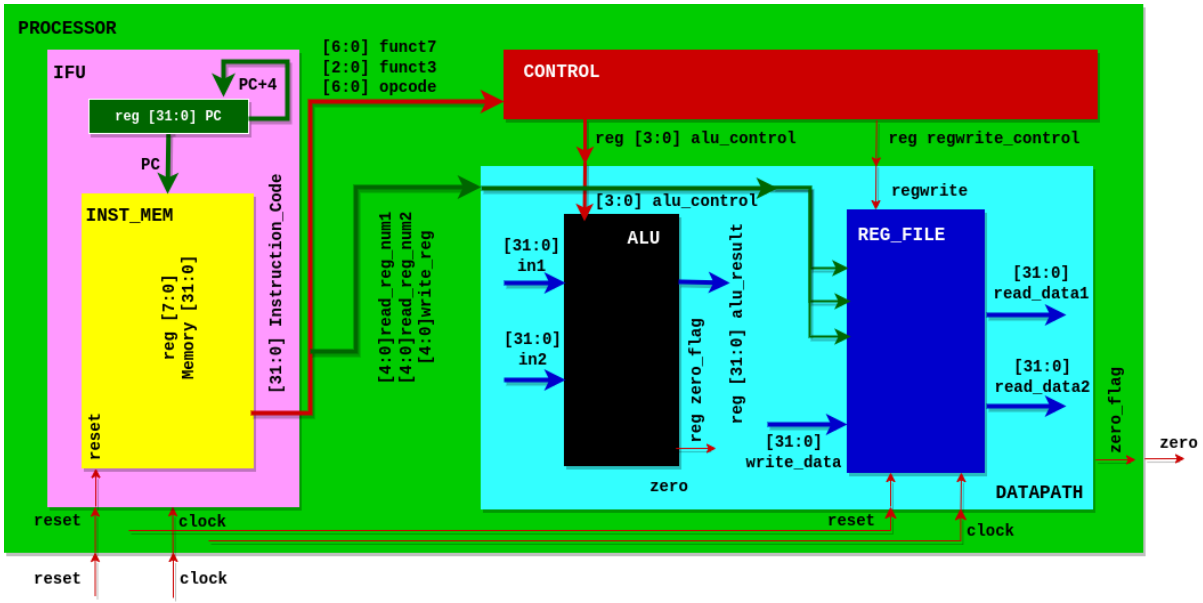
RVI - R-type : addw a5,a5,a4

funct7	rs2	rs1	fun3	rd	opcode
0000000	01110	01111	000	01111	0111011



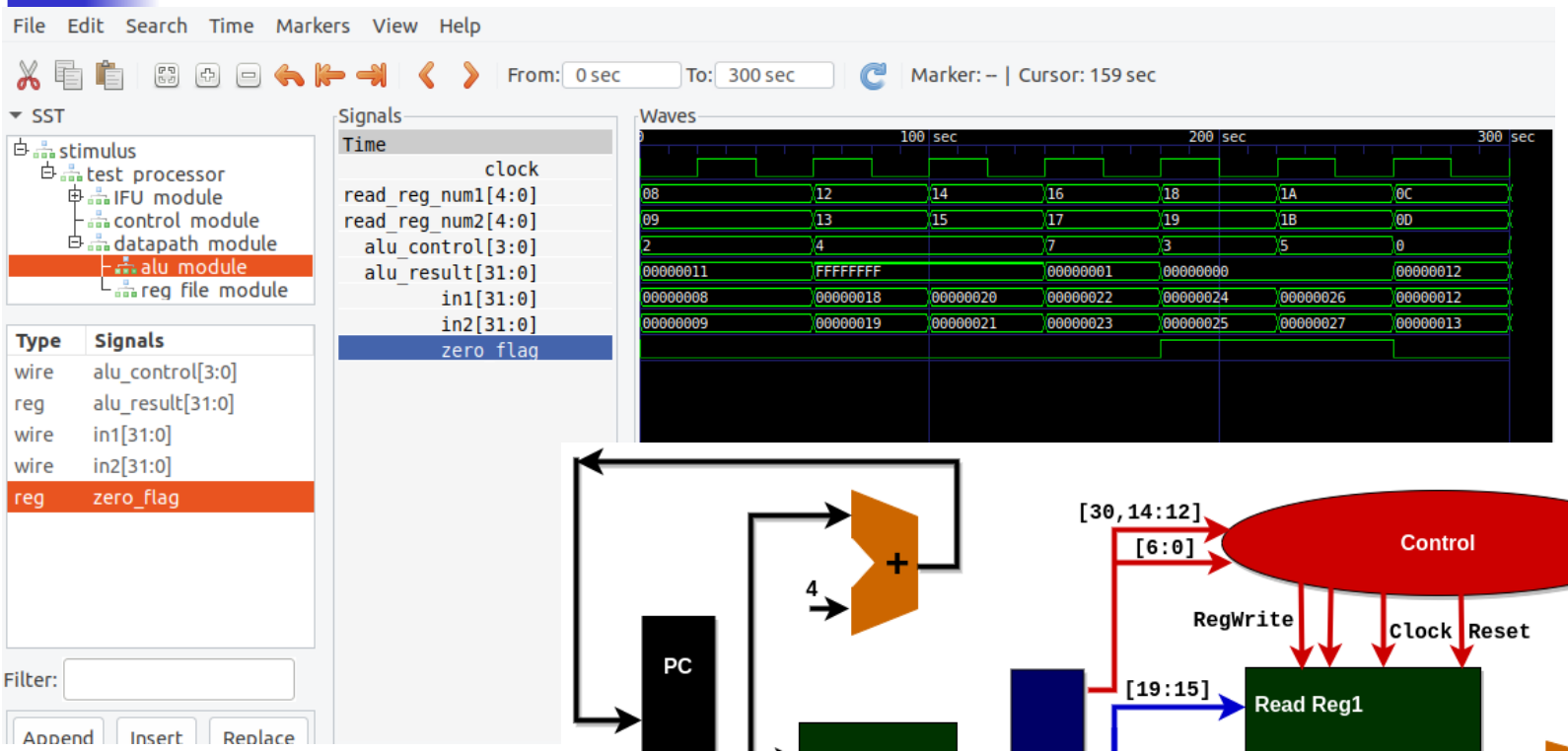


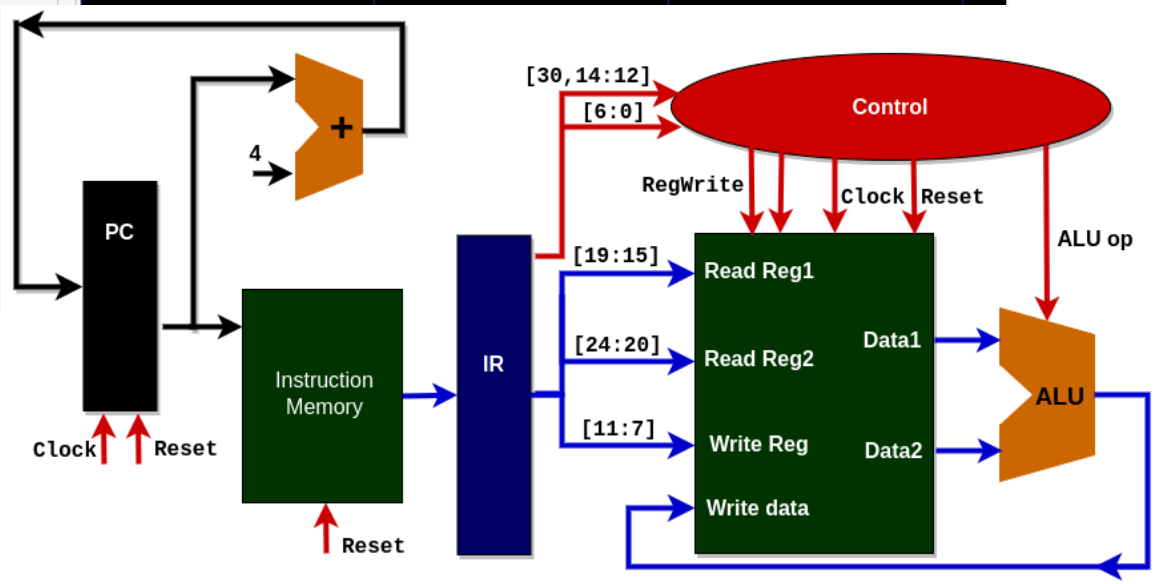
MLab2.1 : PROCESSOR




```
`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
```

MLab2.1 : test & waveforms

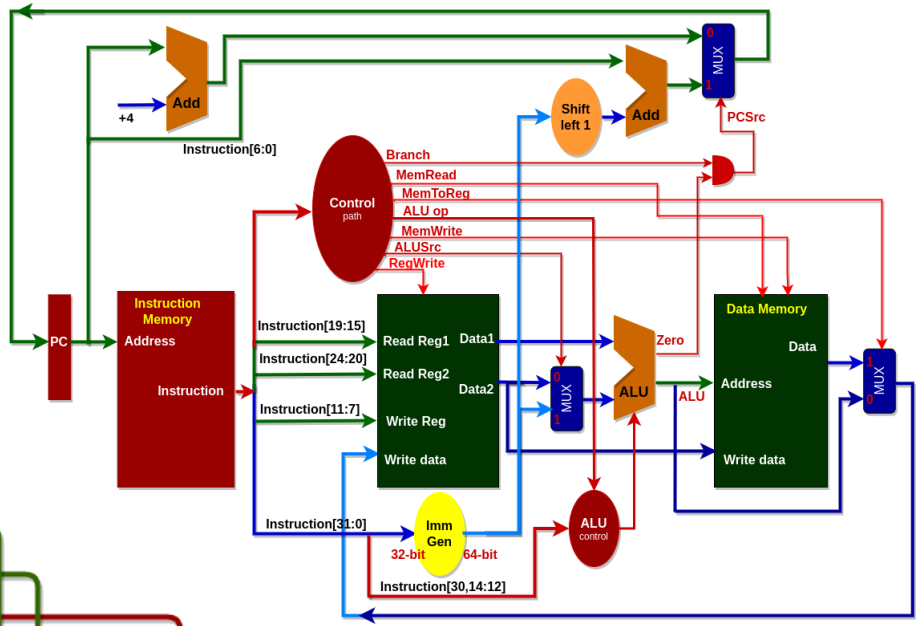


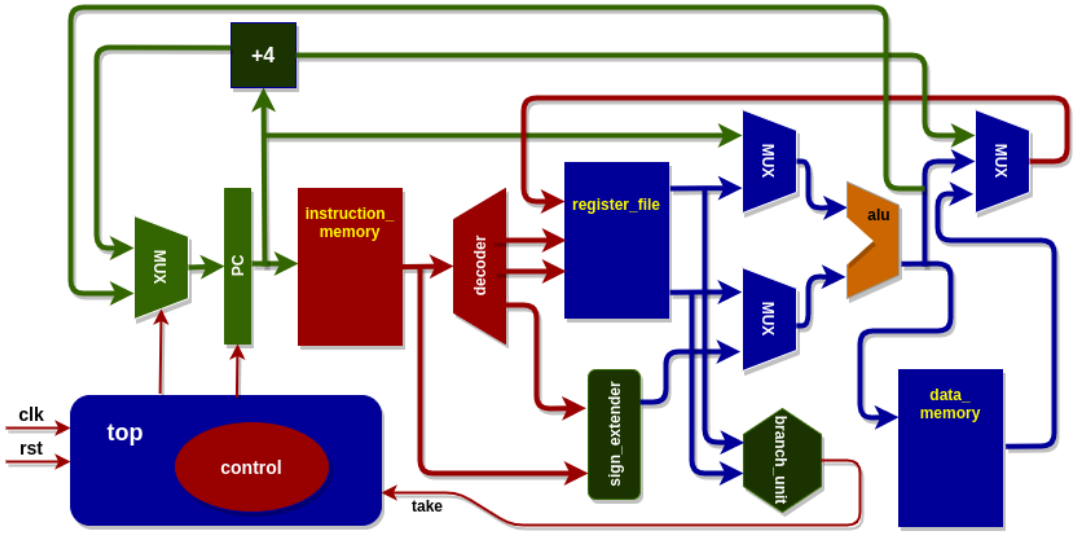


GTKWave waveforms

MLab2.2 : RV32I architecture

RV32I architecture



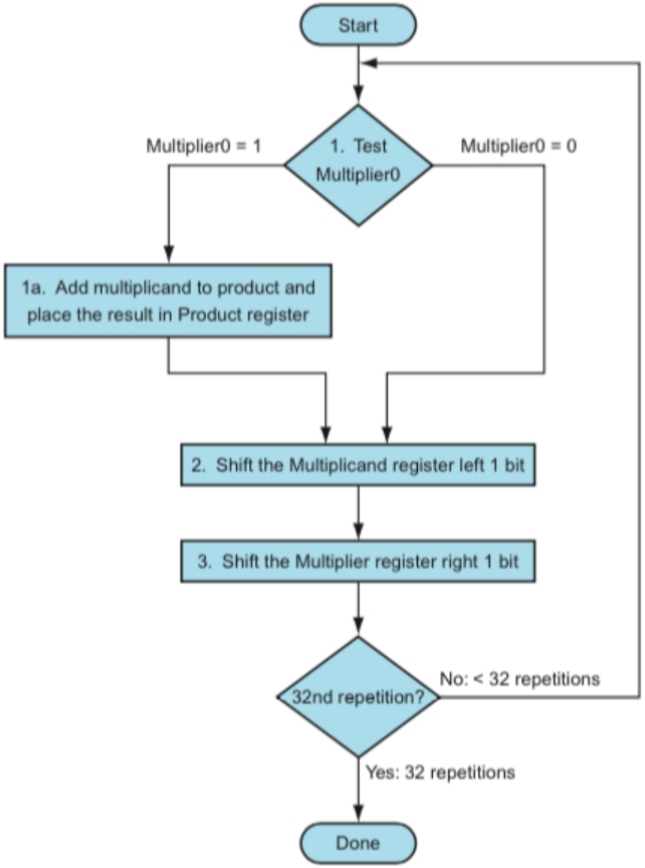


RV32I Verilog model (modules)

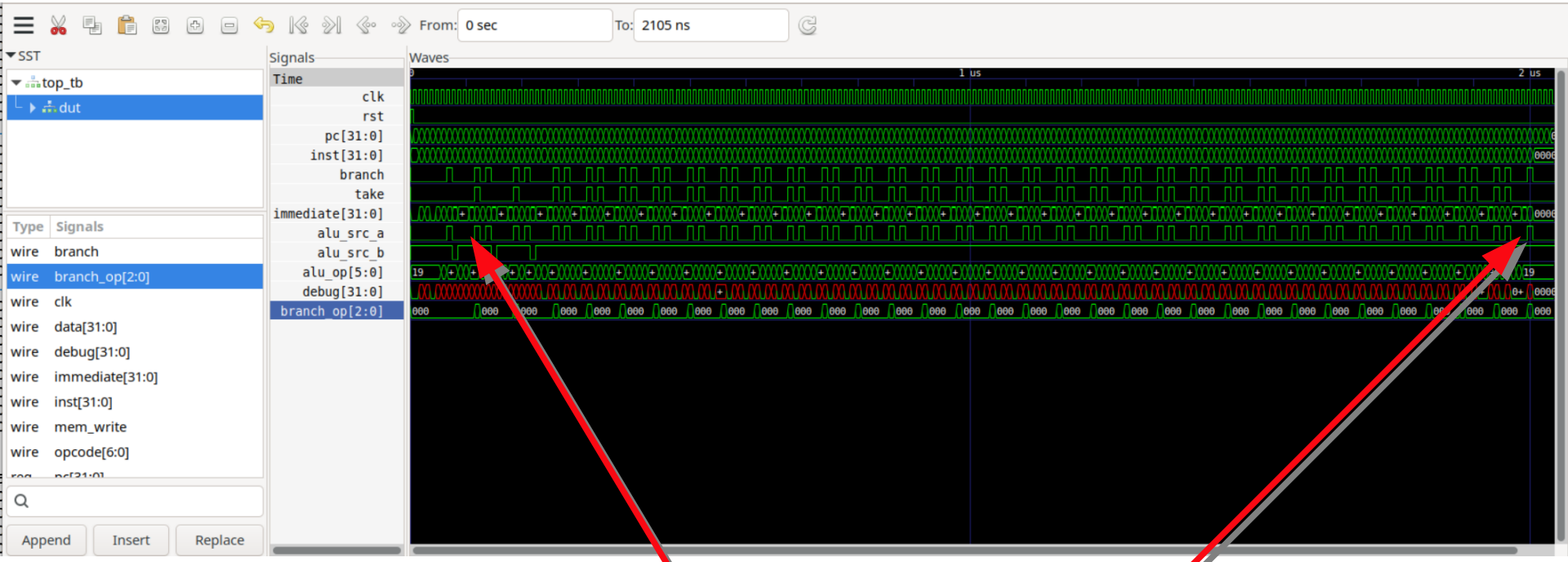
MLab2.2 : binary program & test

The following is our test program elaborated in programming labs PLab2 with the debugging test of macros.

```
// instructions.hex
00000013        // addi x0, x0, 0  -nop
00000013        // addi x0, x0, 0  -nop
00600313        // li   t1,6                       : 0x000
00700393        // li   t2,7                       : 0x004
00000e13        // li   t3,0                       : 0x008
02000593        // li   a1,32                      : 0x00c
0013f293        // andi t0,t2,1                    : 0x010
00028463        // beqz t0,0x101ac <skip_add>      : 0x014
006e0e33        // add  t3,t3,t1                   : 0x018
00131313        // slli t1,t1,0x1                  : 0x01c
0013d393        // srli t2,t2,0x1                  : 0x020
fff58593        // addi a1,a1,-1                   : 0x024
fe0594e3        // bnez a1,0x101a0 <loop>          : 0x028
00000013        // addi x0, x0, 0  -nop
```

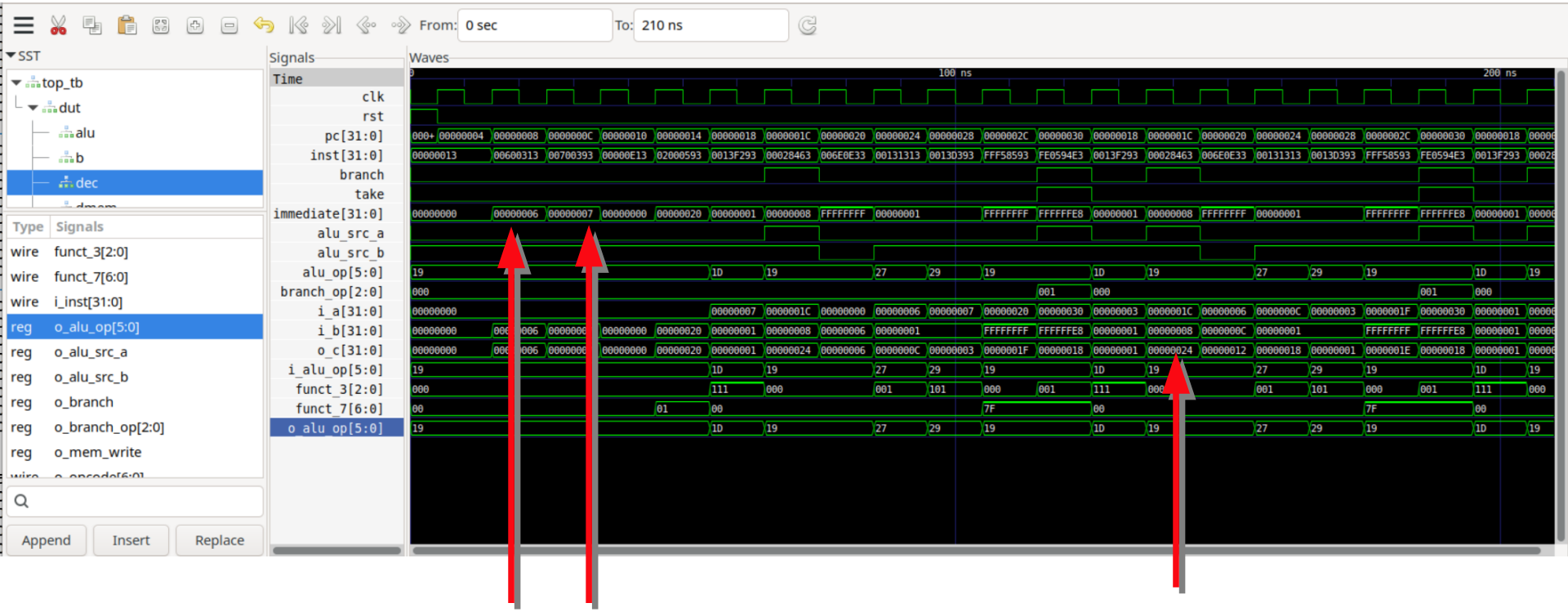


MLab2.2 : test and waveforms



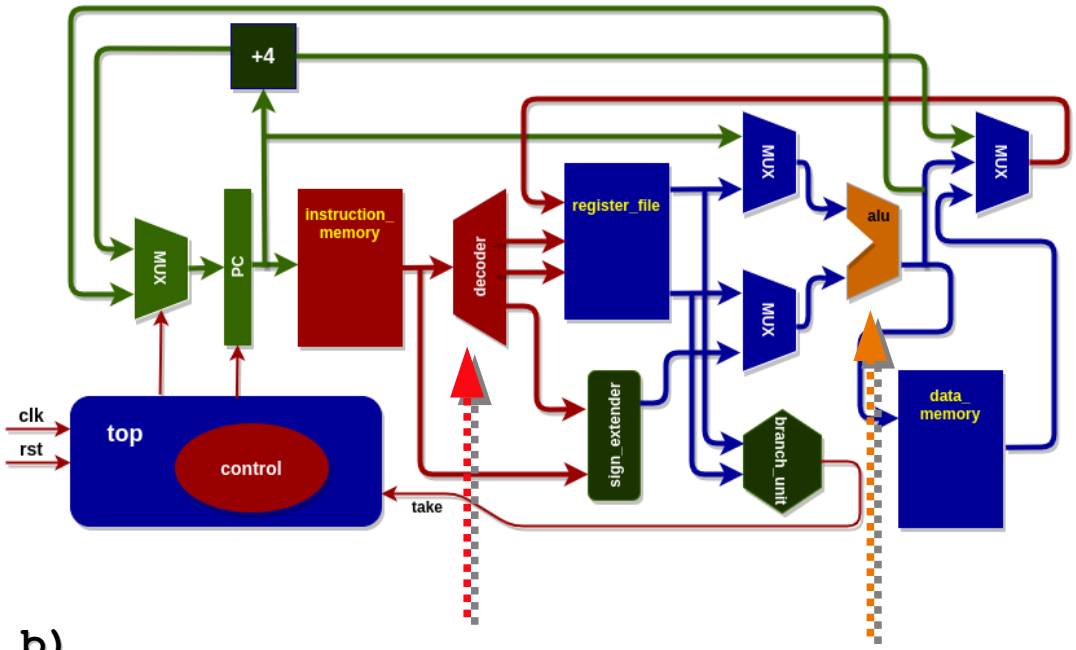
Multiplication cycle: 32 steps

MLab2.2 : test and waveforms



0x06*0x07 = 0x24

MLab2.3 : RV32IM – final project



Final project:

1. Programming assembly function (macro) for `pow(a,b)`

2 Disassembly to binary code

3. Implement on the RV32IM model with multiplication instructions.

Decoding:

`MUL, MULH,`
`DIV, DIVU,`
`REM, REMU`

Extending:

`MUL, MULH,`
`DIV, DIVU,`
`REM, REMU`

MLab2.3 : program & disassemble

```
        .section .data
        result:  .word 0  # Memory space to store the result
        .section .text
        .globl _start
        # Macro Definition: Computes result = base ^ exponent
        .macro power_mult result, base, exponent
            li      \result, 1          # Initialize result with 1
            beqz    \exponent, end_\@   # If exponent is 0, jump to end (result = 1)
        loop_\@:
            mul     \result, \result, \base # result *= base
            addi    \exponent, \exponent, -1 # Decrement exponent
            bnez    \exponent, loop_\@     # Repeat until exponent == 0
        end_\@:
        .endm

    _start:
        # Load base and exponent
        li      t1, 3           # t1 = base = 3
        li      t2, 4           # t2 = exponent = 4
        # Call the power_mult macro: power_mult t3, t1, t2
        power_mult t3, t1, t2
        # Store result in memory
        la      t0, result      # Load address of result in t0
        sw      t3, 0(t0)       # Store t3 (result) to memory
        # Exit the program (using an environment call)
        li      a7, 93          # ECALL for exit
        ecall
```

MLab2.3 : program & disassemble

```
(gdb) b _start
Note: breakpoints 1, 2 and 3 also set at pc 0x100e8.
Breakpoint 4 at 0x100e8: file macro_pow.s, line 22.
(gdb) disassemble /r &_start, +56
Dump of assembler code from 0x100e8 to 0x10120:
   0x00000000000100e8 <_start+0>:   00002197           auipc     gp,0x2
   0x00000000000100ec <_start+4>:   83818193           addi gp,gp,-1992 # 0x11920
   0x00000000000100f0 <_start+8>:   00300313           li    t1,3
   0x00000000000100f4 <_start+12>:  00400393           li    t2,4
   0x00000000000100f8 <_start+16>:  00100e13           li    t3,1
   0x00000000000100fc <_start+20>:  00038863           beqz t2,0x1010c <end_0>
   0x0000000000010100 <loop_0+0>:   026e0e33           mul   t3,t3,t1
   0x0000000000010104 <loop_0+4>:   fff38393           addi t2,t2,-1
   0x0000000000010108 <loop_0+8>:   fe039ce3           bnez t2,0x10100 <loop_0>
   0x000000000001010c <end_0+0>:    00001297           auipc     t0,0x1
   0x0000000000010110 <end_0+4>:    01428293           addi t0,t0,20 # 0x11120
   0x0000000000010114 <end_0+8>:    01c2a023           sw    t3,0(t0)
   0x0000000000010118 <end_0+12>:   05d00893           li    a7,93
   0x000000000001011c <end_0+16>:   00000073           ecall
```

MLab2.3 : load in instruction.hex

Copy the red part of the disassembled code and use it in the `instructions.hex` file loaded into simulated `instruction_memory.v`.

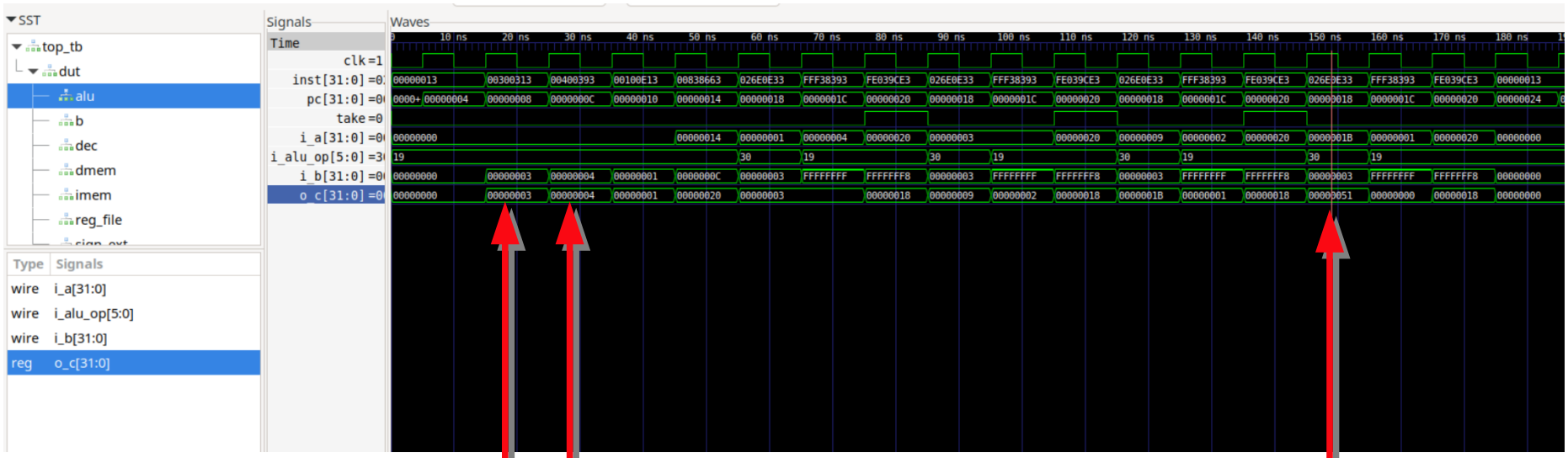
Replace other instructions with `nop` instruction: `00000013`

```
00000013        // addi x0, x0, 0  -nop
00000013        // addi x0, x0, 0  -nop
00300313        // li   t1,3                      : 0x000
00400393        // li   t2,4                      : 0x004
00100e13        // li   t3,1                      : 0x008
00838663        // beq  t2, zero, 0x1010c <end_0>  : bne t0,zero, offset  (4*2)
026e0e33        // mul  t3,t3,t1                  : 0x018
fff38393        // addi t2,t2,-1                  : 0x024
fe039ce3        // bnez t2,0x10100 <loop_0>       : 0x028
00000013        // addi x0, x0, 0  -nop
00000013        // addi x0, x0, 0  -nop
00000013        // addi x0, x0, 0  -nop
```

MLab2.3 : simulate with testbench

```
$ iverilog alu_mult.v branch_unit.v data_memory.v decoder.v
instruction_memory.v register_file.v sign_extension.v top_final.v
tb_top_final.v -o tb_top_final_mult

$ vvp tb_top_final_mult
zero = 00000000  ra = 00000000  sp = 00000000  gp = 00000000
  tp = 00000000  t0 = 00000000  t1 = 00000003  t2 = 00000004
  s0 = 00000000  s1 = 00000000  a0 = 00000000  a1 = 00000000
  a2 = 00000000  a3 = 00000000  a4 = 00000000  a5 = 00000000
  a6 = 00000000  a7 = 00000000  s2 = 00000000  s3 = 00000000
  s4 = 00000000  s5 = 00000000  s6 = 00000000  s7 = 00000000
  s8 = 00000000  s9 = 00000000 s10 = 00000000 s11 = 00000000
  t3 = 00000001  t4 = 00000000  t5 = 00000000  t6 = 00000000
..
..
zero = 00000000  ra = 00000000  sp = 00000000  gp = 00000000
  tp = 00000000  t0 = 00000000  t1 = 00000003  t2 = 00000001
  s0 = 00000000  s1 = 00000000  a0 = 00000000  a1 = 00000000
  a2 = 00000000  a3 = 00000000  a4 = 00000000  a5 = 00000000
  a6 = 00000000  a7 = 00000000  s2 = 00000000  s3 = 00000000
  s4 = 00000000  s5 = 00000000  s6 = 00000000  s7 = 00000000
  s8 = 00000000  s9 = 00000000 s10 = 00000000 s11 = 00000000
  t3 = 00000051  t4 = 00000000  t5 = 00000000  t6 = 00000000
```

MLab2.3 : simulation wave



0x03^0x04 = 0x51

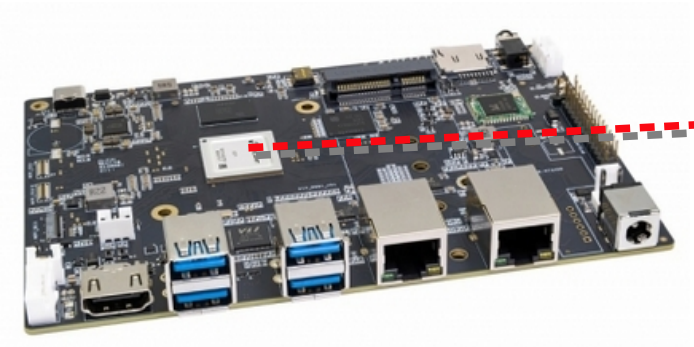
The Platform

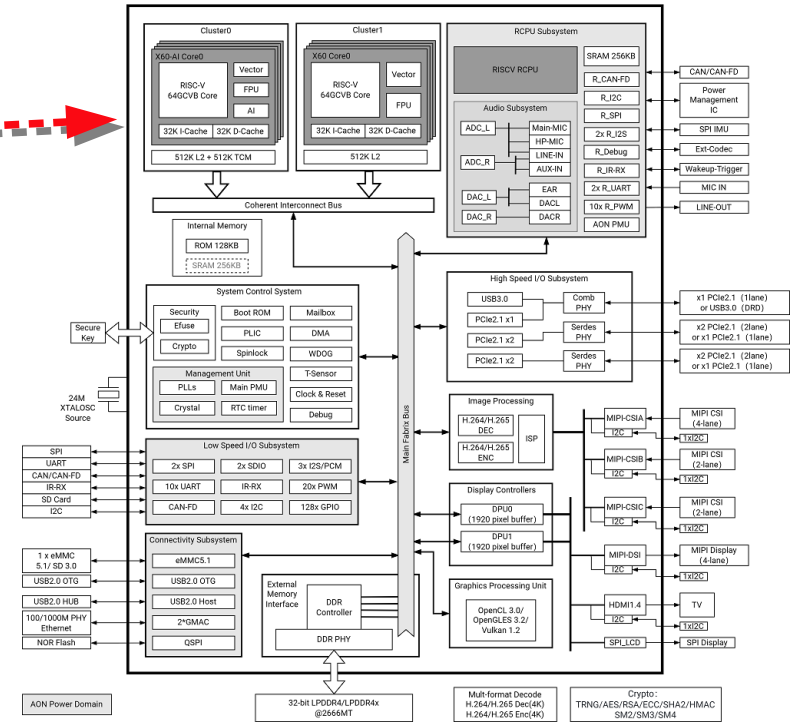
Software:

Ubuntu, C/V-intrinsics/assembly, `gcc`, `riscv64-linux-gnu-objdump`; Verilog, `iverilog`, `GTKwave`, ..

Hardware:

BPI-SF3, SpacemiT K1-X60: RV64GCVB + AI (16 instructions)





4*RV64GCVB + AI

and

4*RV64GCVB

The Platform

Source: Semico Research Corp.

Thank you for your attention !