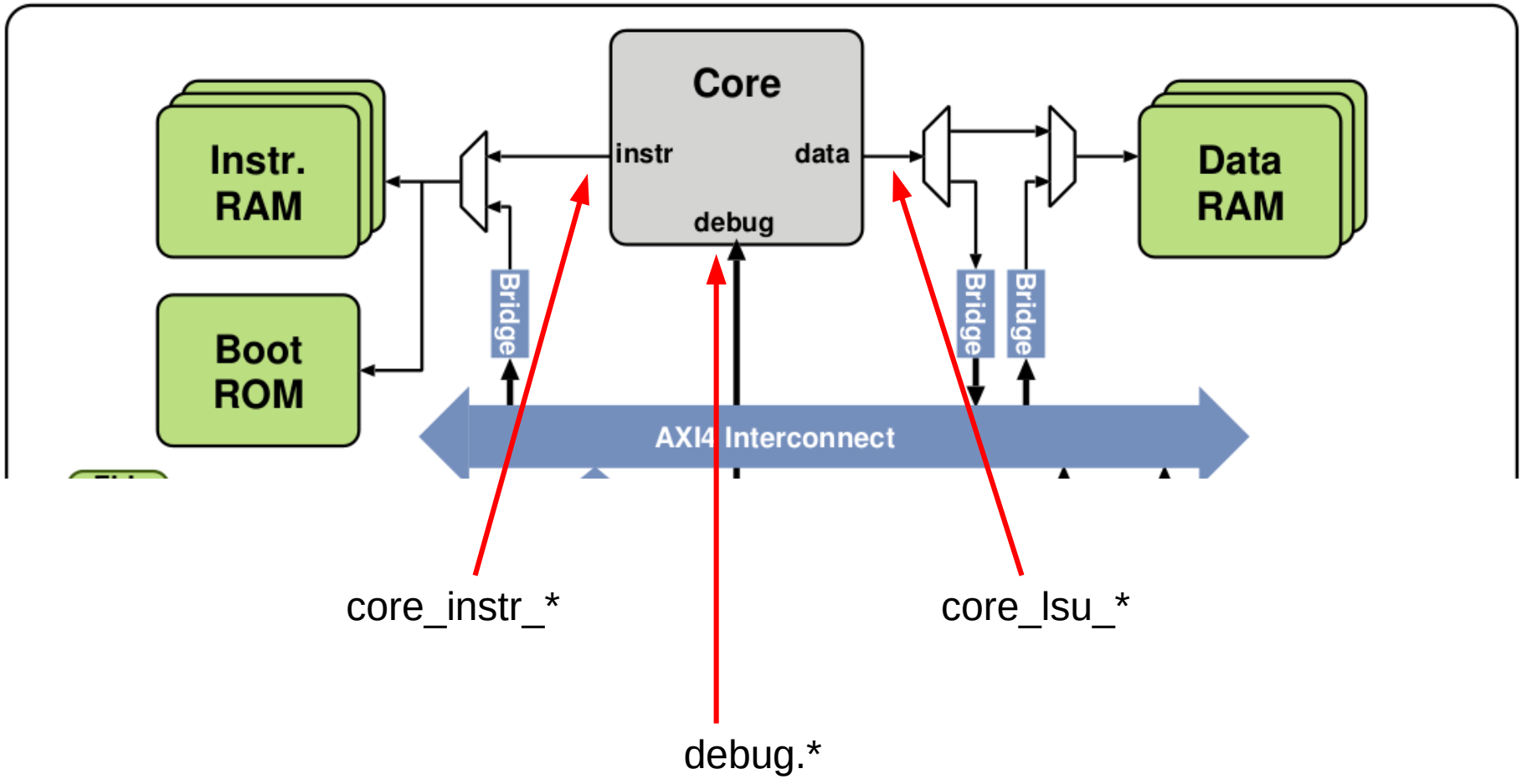
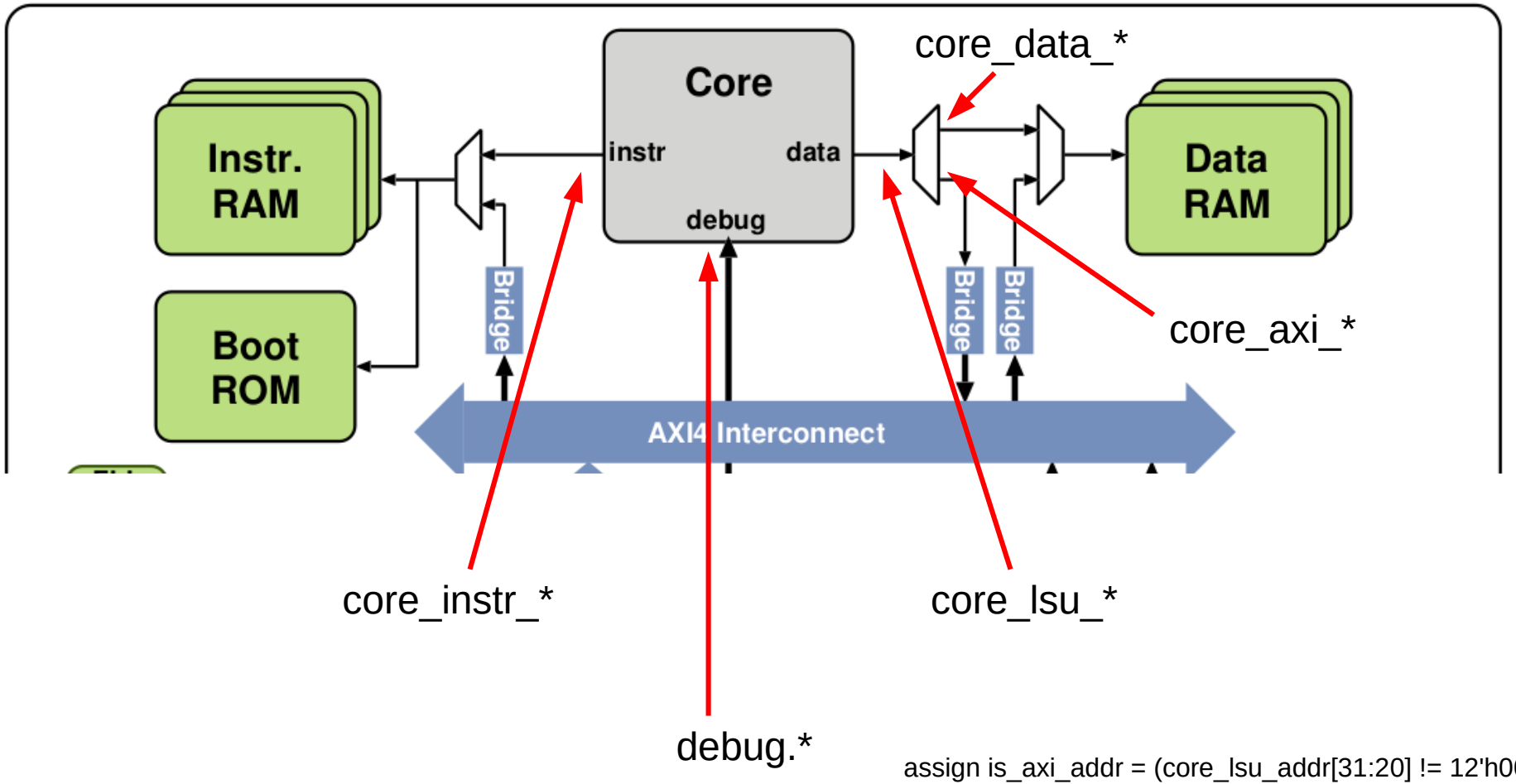


core_region.sv

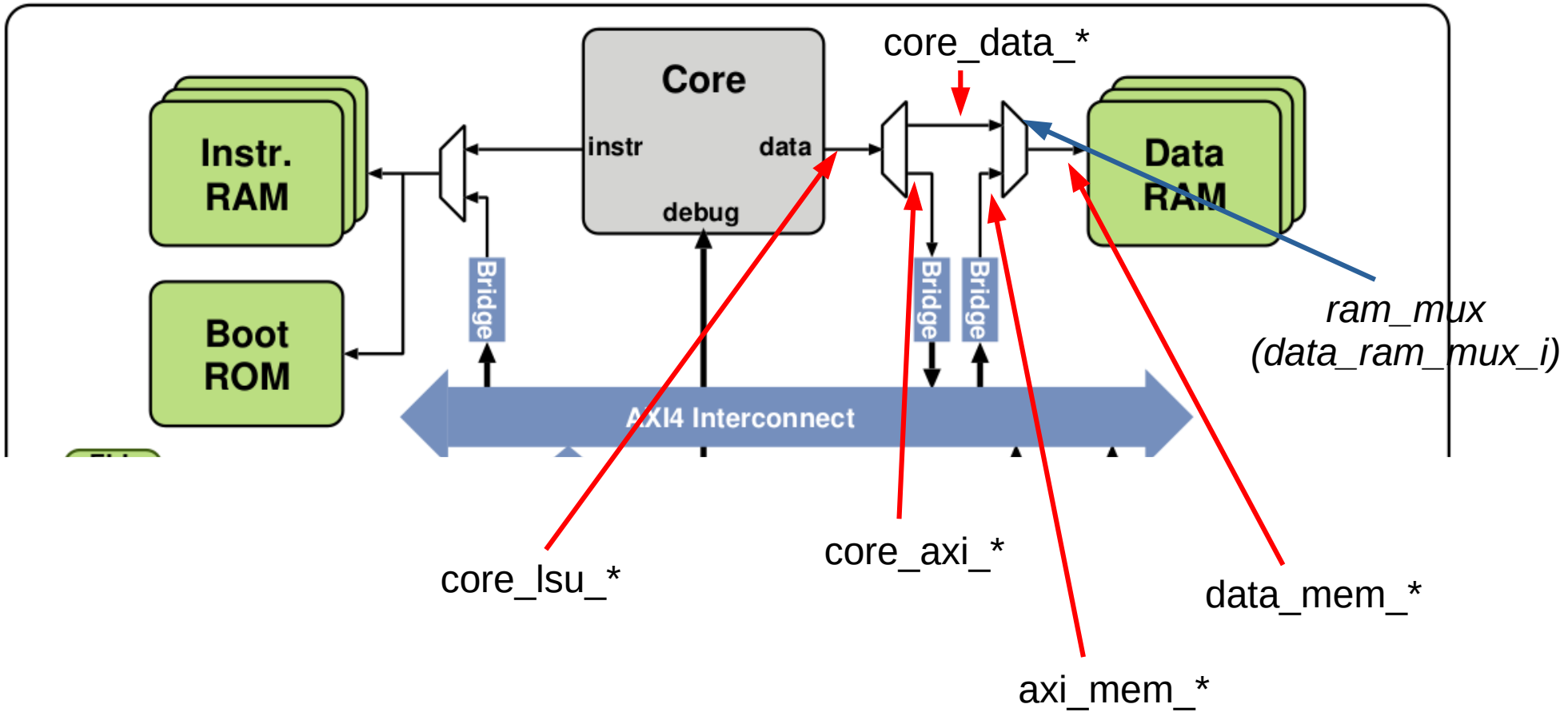


core_region.sv

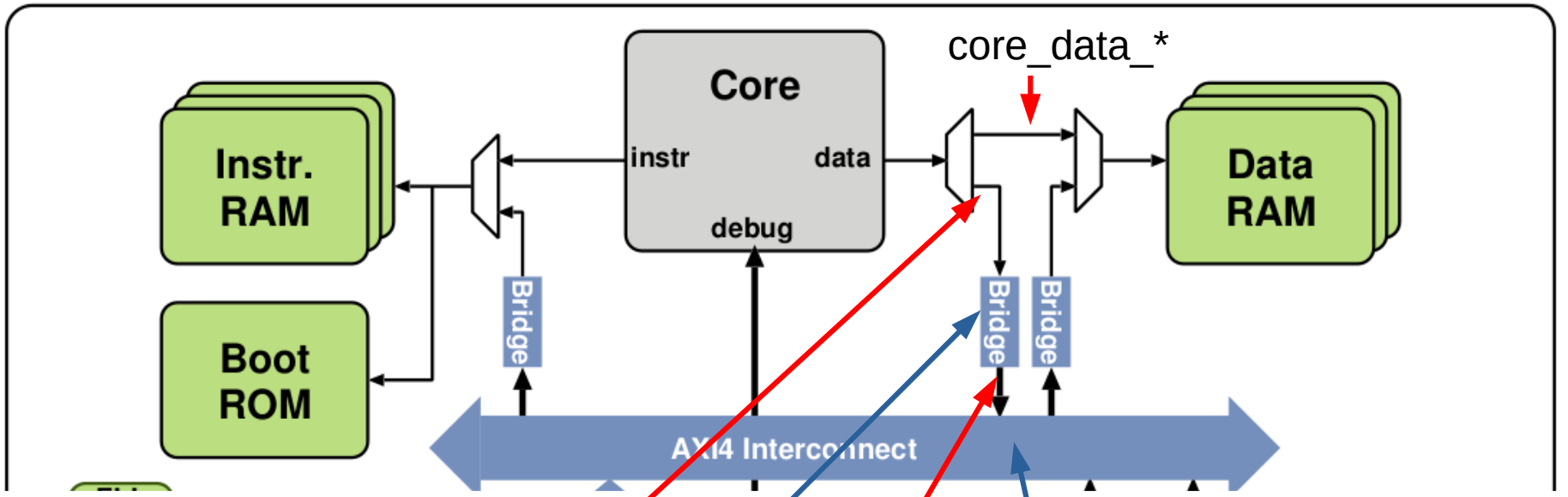
```
assign is_axi_addr = (core_lsu_addr[31:20] != 12'h001);

assign core_data_req = (~is_axi_addr) & core_lsu_req;

assign core_axi_req = is_axi_addr & core_lsu_req;
```



core_region.sv



core_axi_*

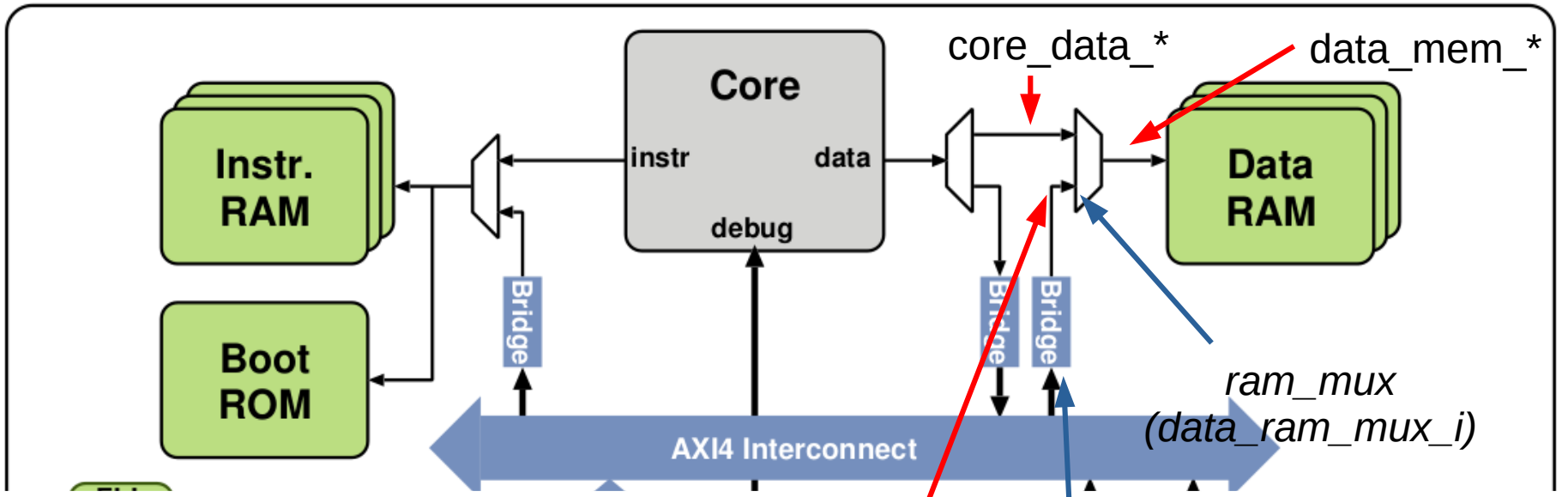
core_master_int

core2axi_wrap
(core2axi_i)

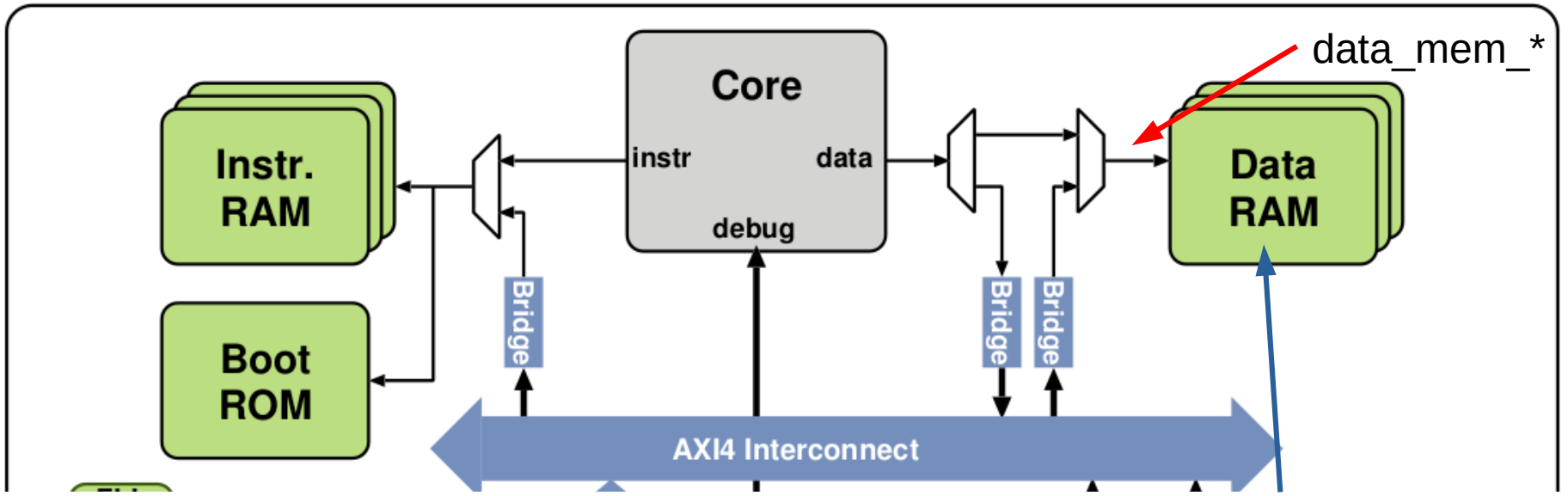
axi_slice_wrap
(axi_slice_core2axi_i)

core_region.sv

CORE_MASTER



core_region.sv



core_region.sv

Netlist generation and simulation #114

Closed **FrischAd** opened this issue on Sep 14, 2017 · 80 comments

FrischAd commented on Sep 14, 2017

Hello,
I generated a netlist using Design Compiler.
I have two issues:

1. In sp_ram_wrap.sv is a module called 'sp_ram_bank' but there is no file containing this module. Where can I find it?
2. I generated the netlist by using the 'sp_ram' and not the 'sp_ram_bank'. Is it possible to simulate this netlist with your provided scripts or alter them easily? If not, how did you verify that the netlist is correct? (Formal verification?)

Thanks in advance.

FrancescoConti commented on Sep 14, 2017 • edited Member

You probably want to replace `sp_ram_bank` with a single ported memory macro instantiation for your technology of choice if you're doing ASIC synthesis. I'm pretty sure the one in the repo is just a placeholder.
Using the functional `sp_ram` which can be found in `rtl/components`, you will end up with a memory composed of flip-flops (standard cells), which is likely not what you want.

If you perform a full-blown synthesis with a "valid" memory macro, you will be able to verify if it works (you can typically also do this at RTL as most macros have functional models). We actually taped out several chips using the code in PULPino and we verified functionality at several levels, including the final product.

If you are wondering *why* we don't distribute memory macros but just a placeholder, there's basically two reasons: first, our distribution is technology-agnostic (at least for what concerns RTL); second, and most important, we cannot distribute memory functional models or macros from real-world foundries as they are not ours. We typically access them under non-disclosure agreements.

OPEN_RAM

```
input  clk0; // clock
input   csb0; // active low chip select
input  web0; // active low write control
input [NUM_WMASKS-1:0]   wmask0; // write mask
input [ADDR_WIDTH-1:0]  addr0;
input [DATA_WIDTH-1:0]  din0;
output [DATA_WIDTH-1:0] dout0;
input  clk1; // clock
input   csb1; // active low chip select
input [ADDR_WIDTH-1:0]  addr1;
output [DATA_WIDTH-1:0] dout1;
```

PULPINO – sp_ram_wrap.sv

```
`elsif ASIC
  // RAM bypass logic
  logic [31:0] ram_out_int;
  // assign rdata_o = (bypass_en_i) ? wdata_i : ram_out_i
  assign rdata_o = ram_out_int;

  sp_ram_bank
  #(
   .NUM_BANKS  ( RAM_SIZE/4096 ),
   .BANK_SIZE  ( 1024          )
  )
  sp_ram_bank_i
  (
   .clk_i   ( clk                  ),
   .rstn_i  ( rstn_i               ),
   .en_i    ( en_i                 ),
   .addr_i  ( addr_i               ),
   .wdata_i ( wdata_i              ),
   .rdata_o ( ram_out_int          ),
   .we_i    ( (we_i & ~bypass_en_i)   ),
   .be_i    ( be_i                 )
  );
```

FLL

```
module clk_rst_gen
(
    input  logic                        clk_i,
    input  logic                        rstn_i,

    input  logic                        clk_sel_i,
    input  logic                        clk_standalone_i,
    input  logic                        testmode_i,
    input  logic                        scan_en_i,
    input  logic                        scan_i,
    output logic                        scan_o,

    input  logic                        fll_req_i,
    input  logic                        fll_wrn_i,
    input  logic              [1:0]     fll_add_i,
    input  logic             [31:0]     fll_data_i,
    output logic                        fll_ack_o,
    output logic             [31:0]     fll_r_data_o,
    output logic                        fll_lock_o,

    output logic                        clk_o,
    output logic                        rstn_o
);
```

```
`ifdef ASIC
  umcL65_LL_FLL
  fll_i
  (
    .FLLCLK     ( clk_fll_int       ),
    .FLLOE      ( 1'b1              ),
    .REFCLK     ( clk_i             ),
    .LOCK       ( fll_lock_o        ),
    .CFGREQ     ( fll_req_i         ),
    .CFGACK     ( fll_ack_o         ),
    .CFGAD      ( fll_add_i         ),
    .CFGD       ( fll_data_i        ),
    .CFGQ       ( fll_r_data_o      ),
    .CFGWEB     ( fll_wrn_i         ),
    .RSTB       ( rstn_i            ),
    .PWDB       ( clk_sel_i         ),
    .STAB       ( clk_standalone_i  ),
    .TM         ( testmode_i        ),
    .TE         ( scan_en_i         ),
    .TD         ( scan_i            ),
    .TQ         ( scan_o            )
  );
`else
  assign fll_ack_o    = fll_req_i;
  assign fll_r_data_o = 1'b0;
  assign fll_lock_o   = 1'b0;
  assign scan_o       = 1'b0;
`endif
```

FLL bypassing

```
39  //  cluster_clock_mux2
40  //  clk_mux_i
41  //  (
42  //      .clk_sel_i ( clk_sel_i     ),
43  //      .clk0_i    ( clk_i         ),
44  //      .clk1_i    ( clk_fll_int   ),
45  //      .clk_o     ( clk_int       )
46  //  );
47  assign clk_int = clk_i;
48
```

core_region.sv

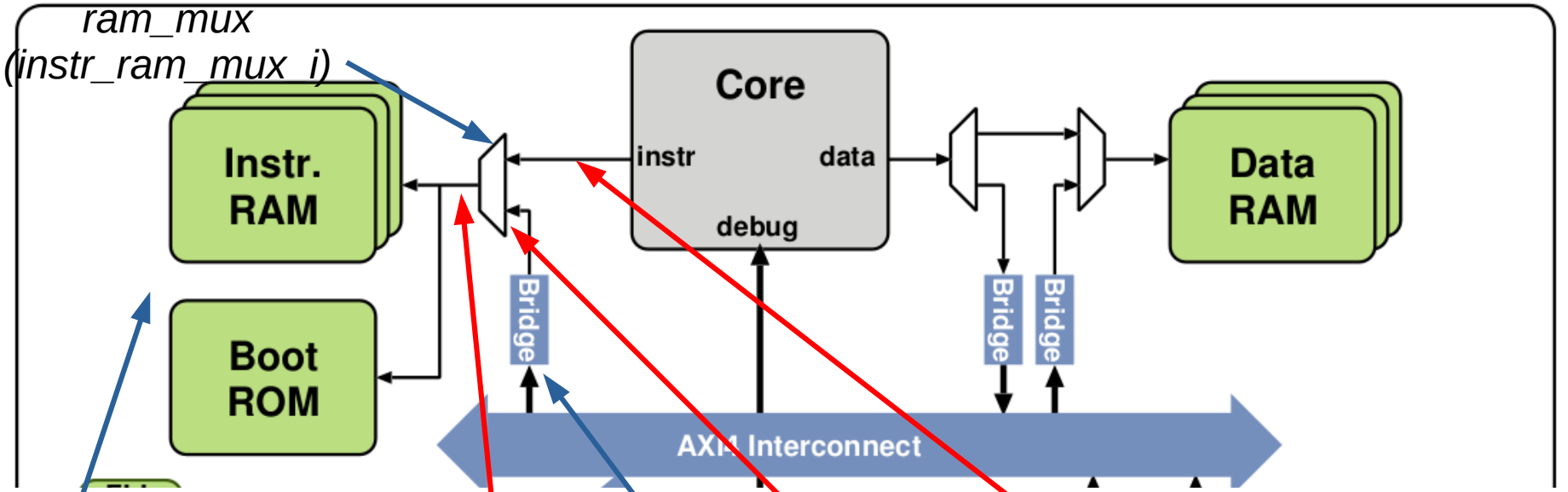
```
53  `ifdef ASIC
54  //  umcL65_LL_FLL
55  //  fll_i
56  //  (
57  //    .FLLCLK     ( clk_fll_int       ),
58  //    .FLLOE      ( 1'b1              ),
59  //    .REFCLK     ( clk_i             ),
60  //    .LOCK       ( fll_lock_o        ),
61  //    .CFGREQ     ( fll_req_i         ),
62  //    .CFGACK     ( fll_ack_o         ),
63  //    .CFGAD      ( fll_add_i         ),
64  //    .CFGD       ( fll_data_i        ),
65  //    .CFGQ       ( fll_r_data_o      ),
66  //    .CFGWEB     ( fll_wrn_i         ),
67  //    .RSTB       ( rstn_i            ),
68  //    .PWDB       ( clk_sel_i         ),
69  //    .STAB       ( clk_standalone_i  ),
70  //    .TM         ( testmode_i        ),
71  //    .TE         ( scan_en_i         ),
72  //    .TD         ( scan_i            ),
73  //    .TQ         ( scan_o            )
74  //  );
75    assign fll_ack_o    = fll_req_i;
76    assign fll_r_data_o = 1'b0;
77    assign fll_lock_o   = 1'b0;
78    assign scan_o       = 1'b0;
79  `else
80    assign fll_ack_o    = fll_req_i;
81    assign fll_r_data_o = 1'b0;
82    assign fll_lock_o   = 1'b0;
83    assign scan_o       = 1'b0;
84  `endif
```

FrancescoConti commented on Oct 17, 2017 • edited

Member

For what concerns the second one, you have to replace `sp_ram_bank` with an instantiation of your own favorite memory cut from your technology library (see the initial question of this same issue).

For what concerns the FLL, this is a clock generator IP that is used in our PULP-based chips (see e.g. http://ieeexplore.ieee.org/abstract/document/7934447/) but is not released as open source as far as I know. If you want to tape out a chip you will of course need a clock generator IP (not necessarily this one), but if you just want to synthesize for comparisons or validation, than you can probably simply bypass it (remove the IP from `clk_rst_gen.sv` + hardwire `clk_sel_i` to 0 so that the internal clock is tied to the reference clock `clk_i`).



ram_mux
(instr_ram_mux_i)

instr_ram_wrap
(instr_mem)

sp_ram_wrap
(sp_ram_wrap_i)

boot_rom_wrap
(boot_rom_wrap_i)

instr_mem_*

core_instr_*

axi_instr_*

INSTR_SLAVE

axi_mem_if_SP_wrap
(instr_mem_axi_if)

core_region.sv

```
`include "config.sv"

module boot_rom_wrap
  #(
    parameter ADDR_WIDTH = `ROM_ADDR_WIDTH,
    parameter DATA_WIDTH = 32
  )(
    // Clock and Reset
    input  logic                    clk,
    input  logic                    rst_n,
    input  logic                    en_i,
    input  logic [ADDR_WIDTH-1:0]   addr_i,
    output logic [DATA_WIDTH-1:0]   rdata_o
  );

  boot_code
  boot_code_i
  (
    .CLK   ( clk                    ),
    .RSTN  ( rst_n                  ),
    .CSN   ( ~en_i                  ),
    .A     ( addr_i[ADDR_WIDTH-1:2] ),
    .Q     ( rdata_o                )
  );
endmodule
```

```
module boot_code
(
    input  logic         CLK,
    input  logic         RSTN,

    input  logic         CSN,
    input  logic [9:0]   A,
    output logic [31:0]  Q
);

  const logic [0:547] [31:0] mem = {
    32'h00000013,
    32'h00000013,
    32'h00000013,
    32'h00000013,
    32'h00000013,
    32'h00000013,
    32'h00000013,
    32'h00000013,
```

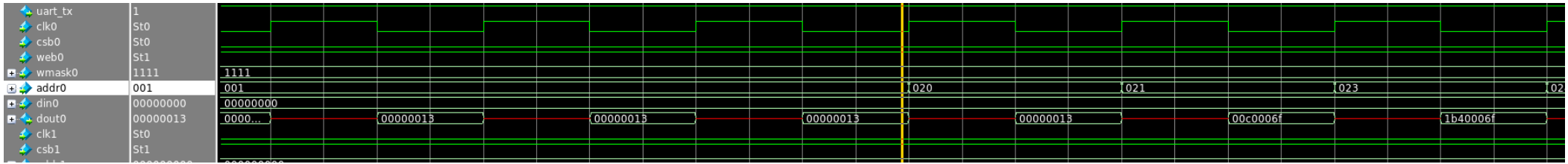
Ram initialization for simulation – DATA MEM

```
51          // preload data memory
52          for(addr = 0; addr < data_size/4; addr = addr) begin
53
54  //          for(bidx = 0; bidx < data_width/8; bidx++) begin
55                  mem_addr = addr / (data_width/32);
56                  data = data_mem[addr];
57
58  //                if (bidx%4 == 0)
59  //                  tb.top_i.core_region_i.data_mem.sp_ram_i.mem[mem_addr][bidx] = data[ 7: 0];
60  //                else if (bidx%4 == 1)
61  //                  tb.top_i.core_region_i.data_mem.sp_ram_i.mem[mem_addr][bidx] = data[15: 8];
62  //                else if (bidx%4 == 2)
63  //                  tb.top_i.core_region_i.data_mem.sp_ram_i.mem[mem_addr][bidx] = data[23:16];
64  //                else if (bidx%4 == 3)
65  //                  tb.top_i.core_region_i.data_mem.sp_ram_i.mem[mem_addr][bidx] = data[31:24];
66  //
67  //                if (bidx%4 == 3) addr++;
68
69                  tb.top_i.core_region_i.data_mem.open_ram_2k.mem[mem_addr] = data[31:0];
70                  addr = addr + 1;
71  //              end
72          end
```

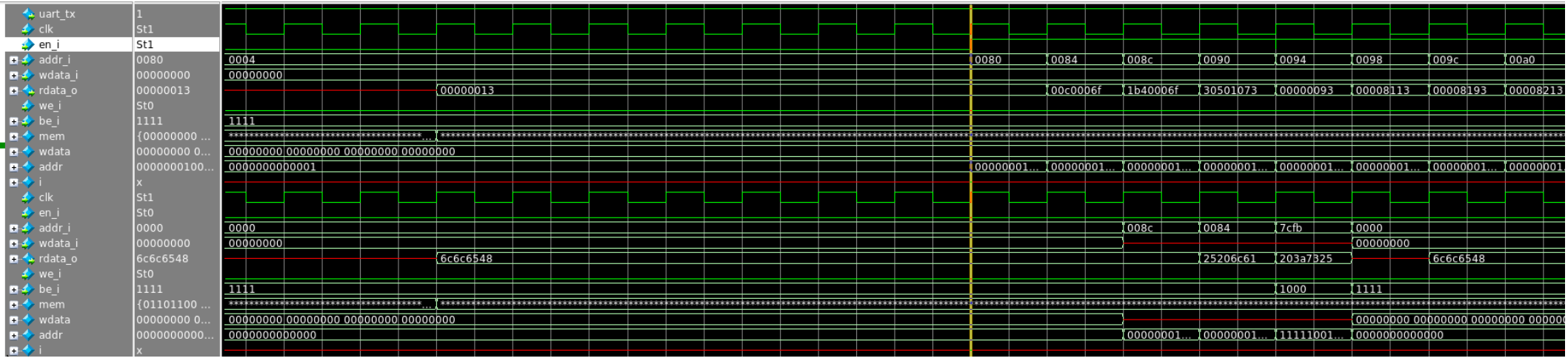
Ram initialization for simulation – INSTR MEM

```
74          // preload instruction memory
75          for(addr = 0; addr < instr_size/4; addr = addr) begin
76
77  //          for(bidx = 0; bidx < instr_width/8; bidx++) begin
78                  mem_addr = addr / (instr_width/32);
79                  data = instr_mem[addr];
80
81  //                if (bidx%4 == 0)
82  //                  tb.top_i.core_region_i.instr_mem.sp_ram_wrap_i.sp_ram_i.mem[mem_addr][bidx] = data[ 7: 0];
83  //                else if (bidx%4 == 1)
84  //                  tb.top_i.core_region_i.instr_mem.sp_ram_wrap_i.sp_ram_i.mem[mem_addr][bidx] = data[15: 8];
85  //                else if (bidx%4 == 2)
86  //                  tb.top_i.core_region_i.instr_mem.sp_ram_wrap_i.sp_ram_i.mem[mem_addr][bidx] = data[23:16];
87  //                else if (bidx%4 == 3)
88  //                  tb.top_i.core_region_i.instr_mem.sp_ram_wrap_i.sp_ram_i.mem[mem_addr][bidx] = data[31:24];
89  //
90  //                if (bidx%4 == 3) addr++;
91
92                  tb.top_i.core_region_i.instr_mem.sp_ram_wrap_i.open_ram_2k.mem[mem_addr] = data[31:0];
93                  addr = addr + 1;
94
95  //          end
96          end
```

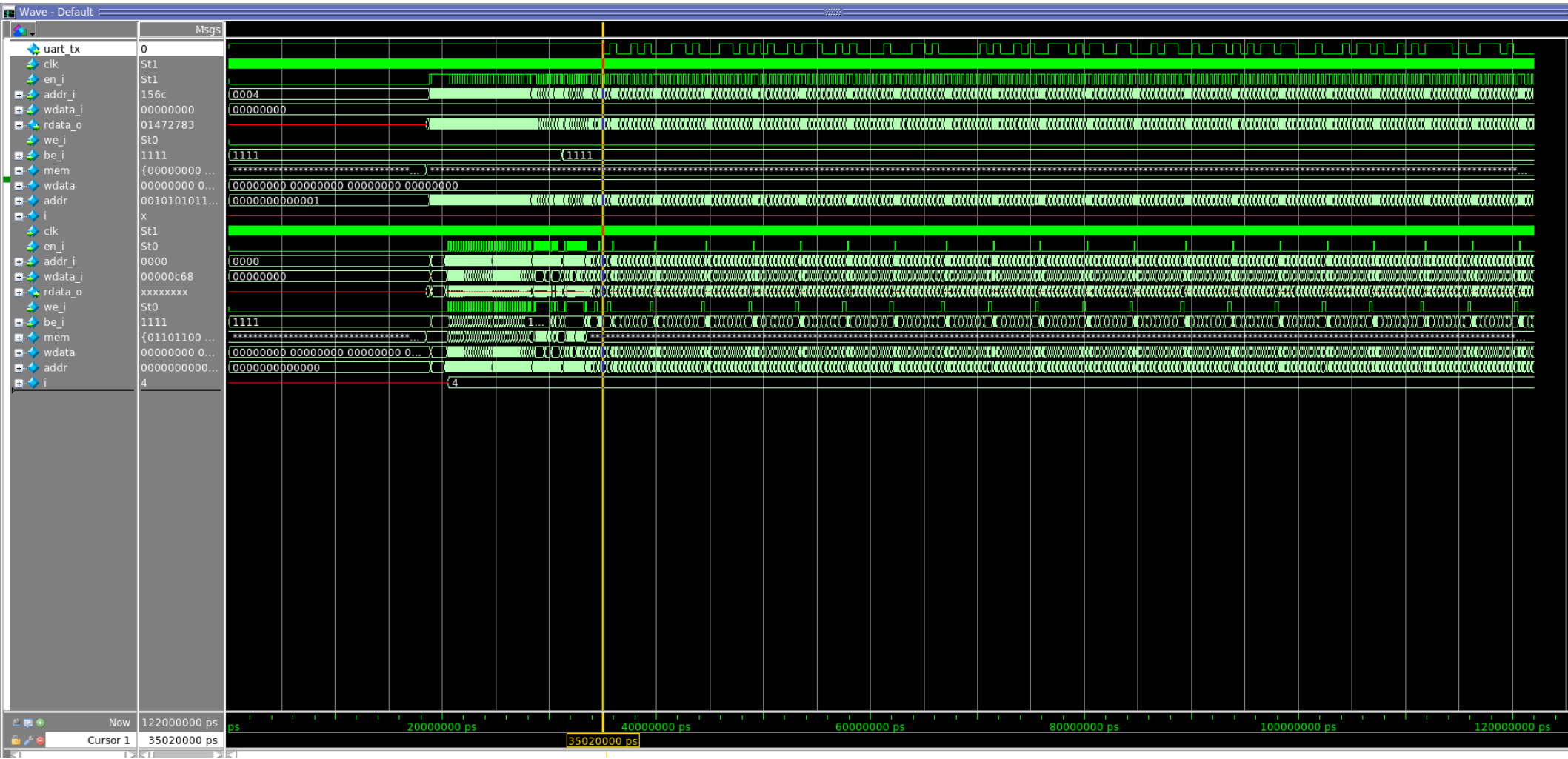
Open ram



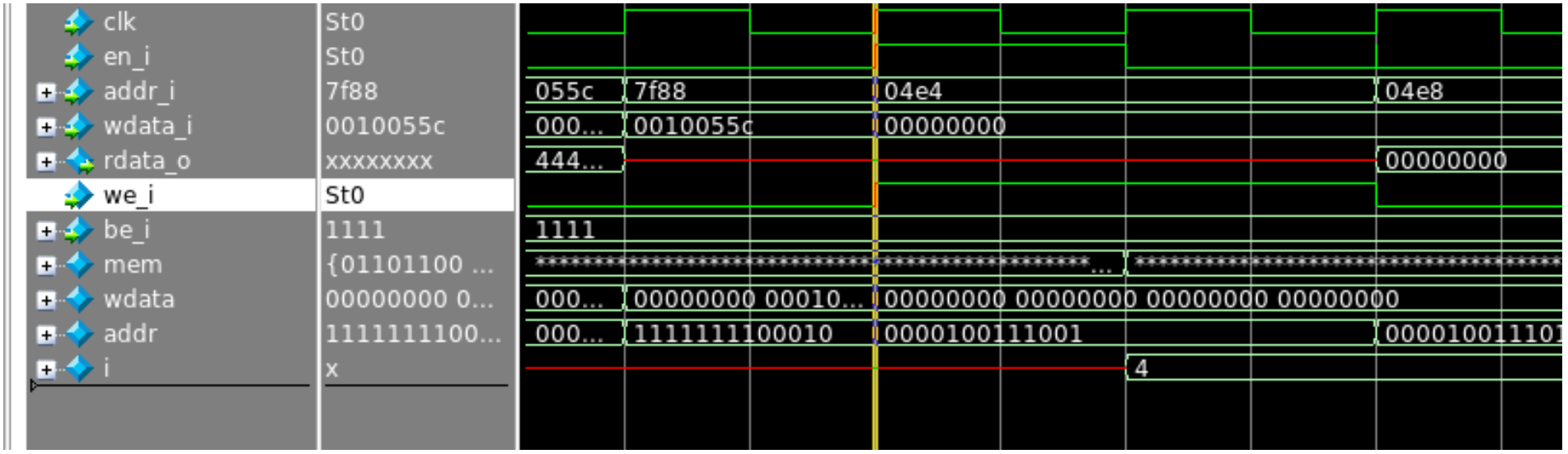
Sp ram



Sp ram

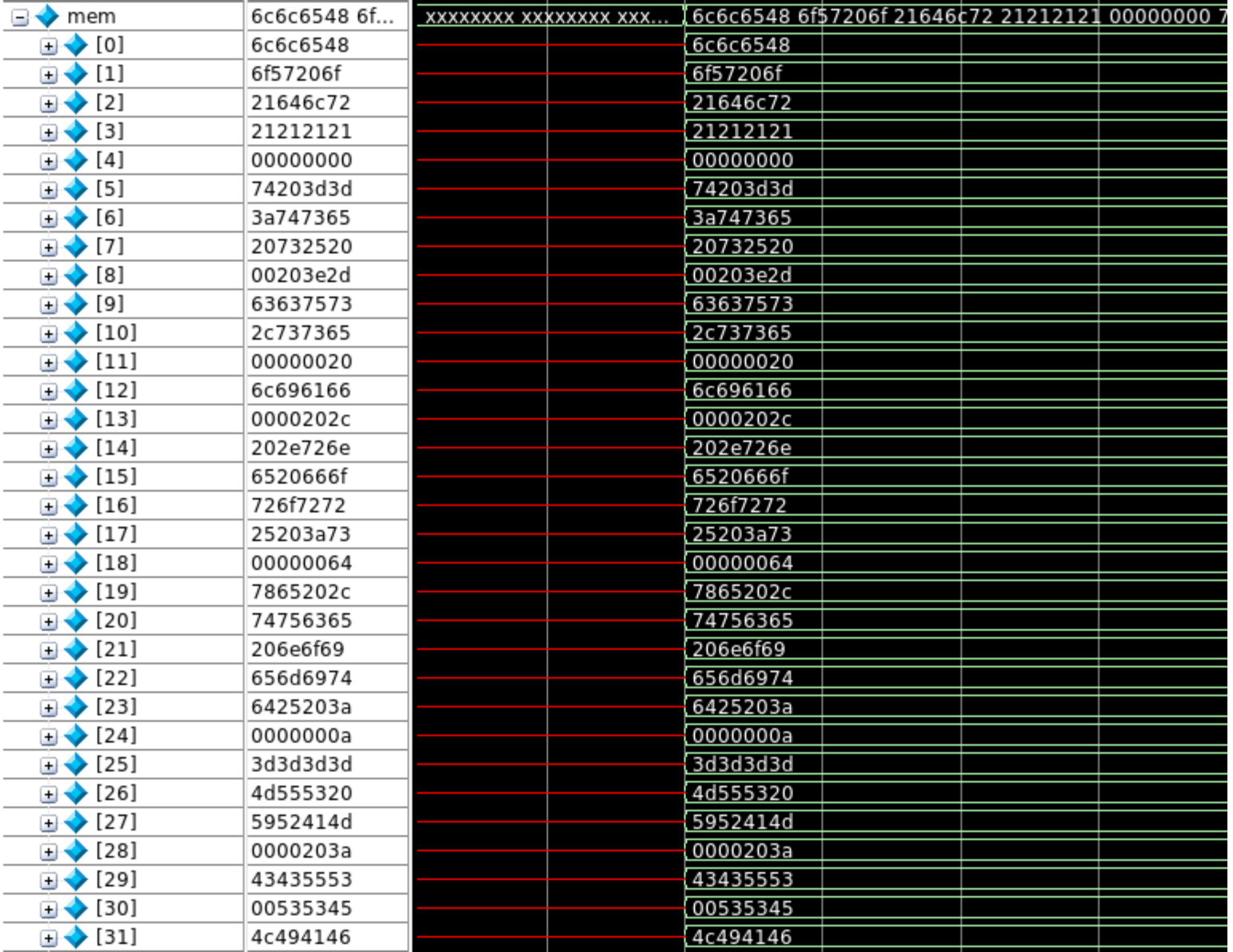

Wave - Default
Msgs
uart_tx 0
clk St1
en_i St1
addr_i 156c 0004
wdata_i 00000000 00000000
rdata_o 01472783
we_i St0
be_i 1111 1111
mem {00000000 ...
wdata 00000000 0... 00000000 00000000 00000000 00000000
addr 0010101011... 0000000000001
i x
clk St1
en_i St0
addr_i 0000 0000
wdata_i 00000c68 00000000
rdata_o xxxxxxxx
we_i St0
be_i 1111 1111 1...
mem {01101100 ...
wdata 00000000 0... 00000000 00000000 00000000 0...
addr 0000000000... 0000000000000
i 4 4
Now 122000000 ps
Cursor 1 35020000 ps
ps
20000000 ps
40000000 ps
60000000 ps
80000000 ps
100000000 ps
120000000 ps
35020000 ps

Sp ram - write


clk
St0
en_i
St0
addr_i
7f88
wdata_i
0010055c
rdata_o
xxxxxxxx
we_i
St0
be_i
1111
mem
{01101100 ...
wdata
00000000 0...
addr
1111111100...
i
x
055c
7f88
04e4
04e8
000...
0010055c
00000000
444...
00000000
1111
000...
00000000 00010...
00000000 00000000 00000000 00000000
000...
1111111100010
0000100111001
4

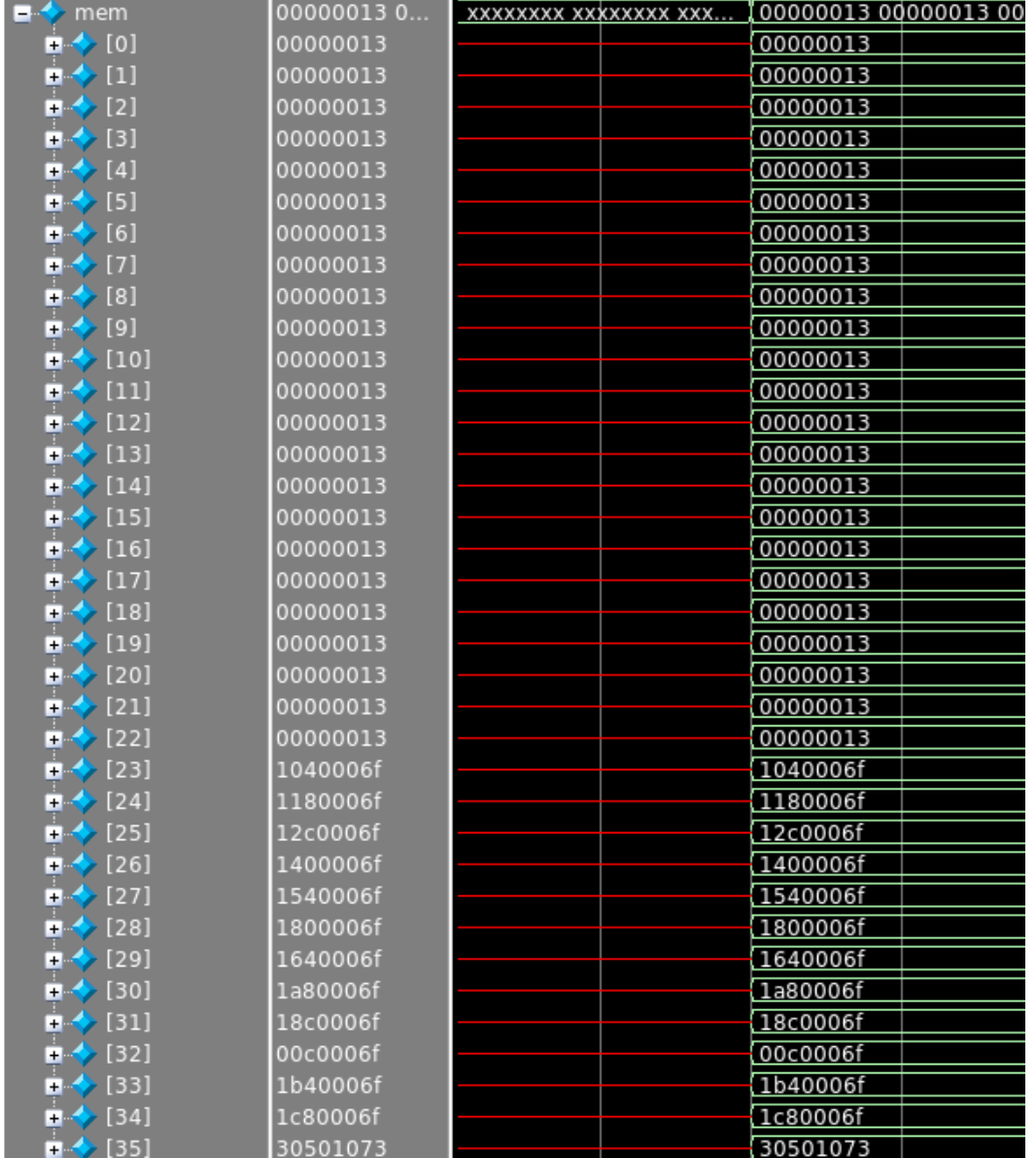
OPEN_RAM

data



OPEN_RAM

instruction



linker

Yazılımsal olarak dataram alanını genişletmek için öncelikle pulpino/sw/ref dizini içerisinde bulunan “link.common.ld” linker dosyası içerisindeki dataram alanı bitişi ve boyutu ve stack’in başlangıcı uygun biçimde ayarlanır. Aynı dizin içerisindeki “link.boot.ld” dosyası içerisinde de stack’in başlangıcı uygun biçimde ayarlanır. Son olarak pulpino/sw/utils dizini içerisindeki “s19toslm.py” dosyası açılır ve “tcdm_bank_size” değişkeni istenilen boyutlarda uygun biçimde değiştirilir. Yazılımsal olarak dataram kısmının genişletilmesiyle ilgili gerekli görseller sırasıyla şekil 3.21, 3.22 ve 3.23’de verilmiştir.

```
MEMORY
{
    instrram    : ORIGIN = 0x00000000, LENGTH = 0x8000
    dataram     : ORIGIN = 0x00100000, LENGTH = 0x1E000
    stack       : ORIGIN = 0x0011E000, LENGTH = 0x2000
}
```

Şekil 3.21: link.common.ld Dosyası İçerisinde Değişim

linker

```
MEMORY
{
    rom         : ORIGIN = 0x00008000, LENGTH = 0x2000
    stack       : ORIGIN = 0x0011E000, LENGTH = 0x2000
}
```

Şekil 3.22: link.boot.ld Dosyası İçerisinde Değişim

```
if(len(sys.argv) < 2):
    print "Usage s19toslm.py FILENAME"
    quit()


l2_banks     = 1
l2_bank_size = 8192 # in words (32 bit)
l2_start     = 0x00000000
l2_end       = l2_start + l2_banks * l2_bank_size * 4 - 1

tcdm_banks     = 1
tcdm_bank_size = 122880 # in words (32 bit)
tcdm_start     = 0x00100000
tcdm_end       = tcdm_start + tcdm_banks * tcdm_bank_size * 4 - 1
tcdm_bank_bits = int(math.log(tcdm_banks, 2))
```

Şekil 3.23: s19toslm.py Dosyası İçerisindeki Değişim

link.common.ld

```
/*
MEMORY
{
    instrram    : ORIGIN = 0x00000000, LENGTH = 0x8000
    dataram     : ORIGIN = 0x00100000, LENGTH = 0x6000
    stack       : ORIGIN = 0x00106000, LENGTH = 0x2000
}*/

MEMORY
{
    instrram    : ORIGIN = 0x00000000, LENGTH = 0x800
    dataram     : ORIGIN = 0x00100000, LENGTH = 0x600
    stack       : ORIGIN = 0x00106000, LENGTH = 0x200
}

/* Stack information variables */
/* _min_stack      = 0x1000; */   /* 4K - minimum stack space to reserve */
_min_stack      = 0x100;   /* 256 - minimum stack space to reserve */
_stack_len     = LENGTH(stack);
_stack_start   = ORIGIN(stack) + LENGTH(stack);
```

s19toslm.py

```
111   l2_banks     = 1
112   #l2_bank_size = 8192 # in words (32 bit)
113   l2_bank_size = 512 # in words (32 bit)
114   l2_start     = 0x00000000
115   l2_end       = l2_start + l2_banks * l2_bank_size * 4 - 1
116
117   tcdm_banks     = 1
118   #tcdm_bank_size = 6144 # in words (32 bit)
119   tcdm_bank_size = 512 # in words (32 bit)
120   tcdm_start     = 0x00100000
121   tcdm_end       = tcdm_start + tcdm_banks * tcdm_bank_size * 4 - 1
122   tcdm_bank_bits = int(math.log(tcdm_banks, 2))
```

Helloworld does not fit In 2K instr mem

make helloworld

/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/5.2.0/../../../../riscv32-unknown-elf/bin/ld: helloworld.elf section `.text.illegal_insn_handler_c' will not fit in region `instrram'
/opt/riscv/bin/../lib/gcc/riscv32-unknown-elf/5.2.0/../../../../riscv32-unknown-elf/bin/ld: region `instrram' overflowed by 3588 bytes

Helloworld does not fit In 2K instr mem

Do not use printf() !!!

I will create a new application what requires less memory

Why I don’t like printf()

How many times have I seen this:

```
1 | printf("Hello world!");
```

I have a strong opinion, and a rule for using `printf()`:

> *“Do *not* use printf()!”*

Using `printf()`, as the other ‘related’ functions like `scanf()`, can be very handy: it is easy to write something to the console, or to build a string. I used this in “printf() with the FRDM-KL25Z Board and without Processor Expert” too ;-). But in general it is bad. Really bad.

Code Size

Using `printf()` adds greatly to the code size of the application. I have seen cases where this is in the range of 10-20 KByte of code. The problem comes from the fact that `printf()`, as defined by the ANSI library standard, needs to support all the different format string. Including formatting octal numbers, floating point, etc. Even if you are not using octal numbers. Or when was the last time I used octal numbers? It is in there, and it adds up.

You might have a look at the `printf()` implementation of your library. If you have CodeWarrior, then have a look at

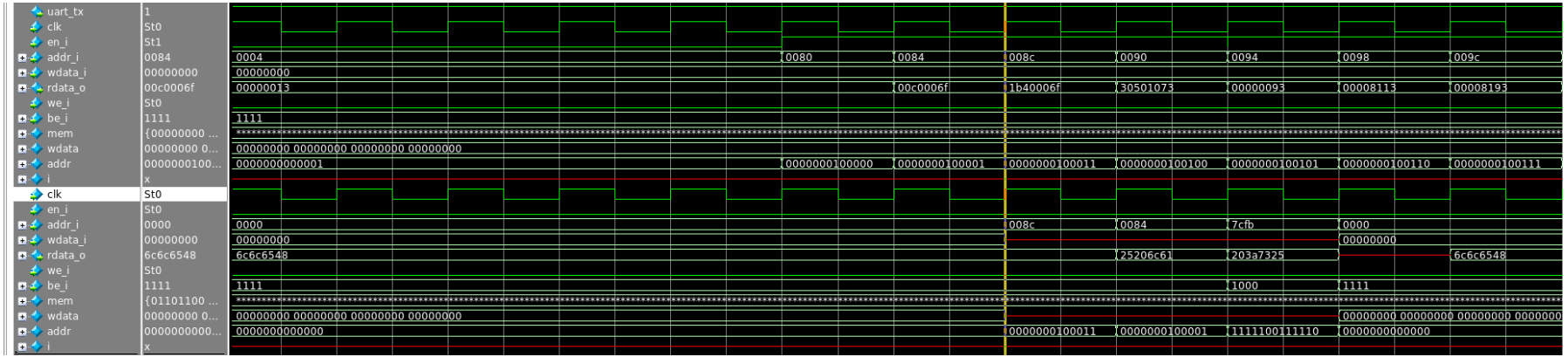
```
\lib\hc08c\src\printf.c
```

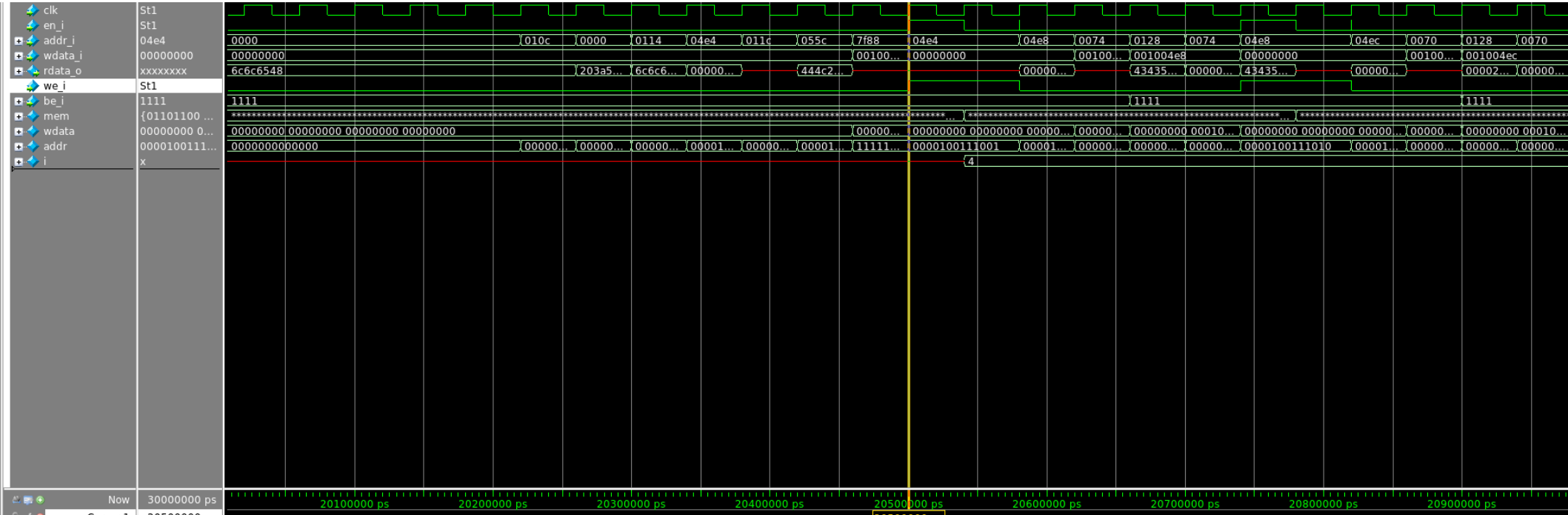
Helloworld fit in 2K by changing compiler flags
-0s and RVC=1

But there are still errors

After some time, core tries to get memory and
instruction from invalid addresses

sp_ram


uart_tx 1
clk St0
en_i St1
addr_i 0084
wdata_i 00000000
rdata_o 00c0006f
we_i St0
be_i 1111
mem {00000000 ...
wdata 00000000 0...
addr 0000000100...
i x
clk St0
en_i St0
addr_i 0000
wdata_i 00000000
rdata_o 6c6c6548
we_i St0
be_i 1111
mem {01101100 ...
wdata 00000000 0...
addr 0000000000...
i x


clk St1
en_i St1
addr_i 04e4
wdata_i 00000000
rdata_o xxxxxxxx
we_i St1
be_i 1111
mem {01101100 ...
wdata 00000000 0...
addr 0000100111...
i x
Now 30000000 ps
20100000 ps
20200000 ps
20300000 ps
20400000 ps
20500000 ps
20600000 ps
20700000 ps
20800000 ps
20900000 ps

Finally it worked !

We need to create a new build folder for each trial change in scripts