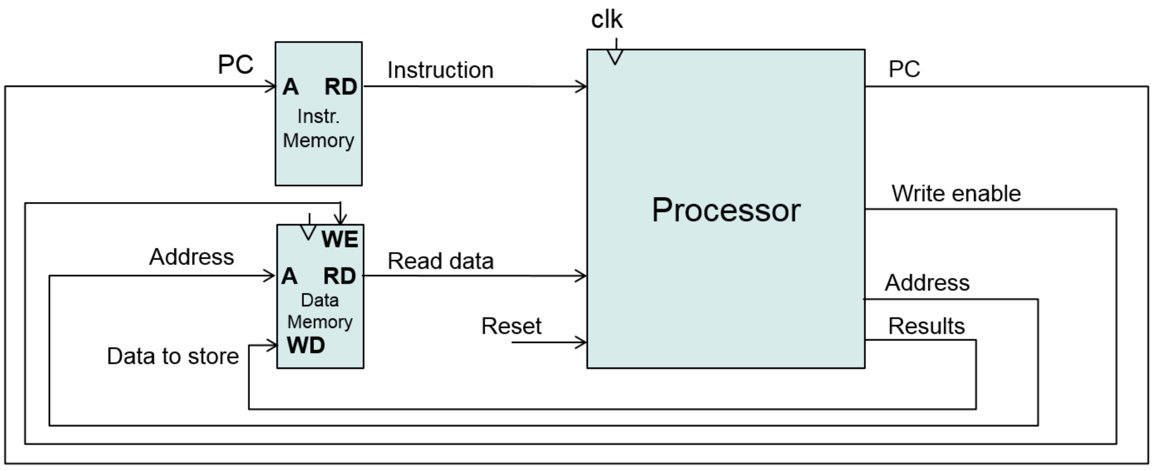
Project #1: Single-cycle processor design

Project #1: Single-cycle processor design

Basic ISA design

Design a simple 32-bit processor connected to a separate instruction and data memory. The processor has to implement instructions given in the table below. Suppose that the processor starts the execution from the beginning of instruction memory (0x00000000).



Instrukce	Syntax	Operace	Poznámka
add	add rd, rs1, rs2	rd ← [rs1] + [rs2];	
addi	addi rd, rs1, $imm_{11:0}$	rd ← [rs1] + $imm_{11:0}$;	
and	and rd, rs1, rs2	rd ← [rs1] & [rs2];	
sub	sub rd, rs1, rs2	rd ← [rs1] - [rs2];	
slt	slt rd, rs1, rs2	if [rs1] < [rs2] then rd←1; else rd←0;	
div	div rd, rs1, rs2	rd ← [rs1] / [rs2];	
rem	rem rd, rs1, rs2	rd ← [rs1] % [rs2];	
beq	beq rs1, rs2, $imm_{12:1}$	if [rs1] == [rs2] go to [PC]+{$imm_{12:1}$,'0'}; else go to [PC]+4;	
blt	blt rs1, rs2, $imm_{12:1}$	if [rs1] < [rs2] go to [PC]+{$imm_{12:1}$,'0'}; else go to [PC]+4;	
lw	lw rd,$imm_{11:0}$(rs1)	rd ← Memory[[rs1] + $imm_{11:0}$]	
sw	sw rs2,$imm_{11:0}$(rs1)	Memory[[rs1] + $imm_{11:0}$] ← [rs2];	
lui	lui rd, $imm_{31:12}$	rd ← {$imm_{31:12}$,'0000 0000 0000'};	
jal	jal rd, $imm_{20:1}$	rd ← [PC]+4; go to [PC] +{$imm_{20:1}$,'0'};	
jalr	jalr rd, rs1, $imm_{11:0}$	rd ← [PC]+4; go to [rs1]+$imm_{11:0}$;	

Extended ISA design

Add to the processor's ISA the following 4 instructions: auipc, sll, srl, sra.

Instruction	Syntax	Operation	Note
auipc	auipc rd,imm$_{31:12}$	rd ← [PC] + {imm$_{31:12}$,'0000 0000 0000'};	
sll	sll rd, rs1, rs2	rd ← [rs1] << [rs2];	
srl	srl rd, rs1, rs2	rd ← (unsigned)[rs1] >> [rs2];	
sra	sra rd, rs1, rs2	rd ← (signed)[rs1] >> [rs2];	

Note: The submission system tests these 4 instructions sequentially (in some order). If it finds an incorrect implementation, it does not continue in testing and it only displays the total number of points for basic and extended ISA design. So, the submission system no longer provides hints.

Instruction encoding:

Each instruction is encoded in 32 bits (in the table from msb towards lsb), where rs1, rs2 and rd are encoded in 5 bits. The last column of the table represents Opcode of the corresponding instruction.

add:	0000000	rs2	rs1	000	rd	0110011
addi	imm[11:0]		rs1	000	rd	0010011
and:	0000000	rs2	rs1	111	rd	0110011
sub:	0100000	rs2	rs1	000	rd	0110011
slt:	0000000	rs2	rs1	010	rd	0110011
div:	0000001	rs2	rs1	100	rd	0110011
rem:	0000001	rs2	rs1	110	rd	0110011
beq:	imm[12|10:5]	rs2	rs1	000	imm[4:1|11]	1100011
blt:	imm[12|10:5]	rs2	rs1	100	imm[4:1|11]	1100011
lw:	imm[11:0]		rs1	010	rd	0000011
sw:	imm[11:5]	rs2	rs1	010	imm[4:0]	0100011
lui:	imm[31:12]				rd	0110111
jal:	imm[20|10:1|11|19:12]				rd	1101111
jalr:	imm[11:0]		rs1	000	rd	1100111
auipc:	imm[31:12]				rd	0010111
sll:	0000000	rs2	rs1	001	rd	0110011
srl:	0000000	rs2	rs1	101	rd	0110011
sra:	0100000	rs2	rs1	101	rd	0110011

Program S

Write program S that will iterate through an array of numbers and determine if the given number is a prime. If there is a prime number in the given position in the array, it overwrites this value with 1, otherwise with 0. The program must repeatedly **call** the **prime** subroutine that accepts 1 argument: the tested number. In C, a function with the following prototype would correspond to this subroutine:

```
int prime(unsigned int number);
```

The subroutine returns 1 if "number" is a prime number. Consider 2 as the smallest prime number. Use the RISC-V calling convention.

Assume that information about the size of the array and its starting address are fixed at addresses in data memory

- 0x00000004: array size (number of items)
- 0x00000008: pointer to starting address of the array

That is, reading the value stored at address 0x00000008 will give you the address where the first element of the array starts.

The program can only modify the array itself, and any modification of the array is considered an answer to the primeness test, i.e. do not put auxiliary data in the array.

You can assume that "number" is always less than 1000. The size of the instruction memory is limited to 128 words, i.e. 512 B. Consequence: a program exceeding this limit will not be accepted.

If you receive 0 points for the program, it may also be an error in the description of the CPU. Full marks for CPU only means that you have demonstrated enough knowledge of Verilog to get full marks for this section (CPU description may be still incomplete or with minor issues).

Seminar project evaluation

Description	Points
Basic CPU design in Verilog	12
Extended CPU design:	4
Program S in the machine code	9

Soft deadline: 8th week (see Evaluation (../classification/index.html)). Each week delay is sanctioned with -2 points. Submissions after the 13th week of the semester are not accepted (hard deadline). You are allowed to upload your solution up to 10 times.

Your program has to run on your CPU design, i.e. you can get 9 points for your program only if you provide your description of the CPU in Verilog.

The requirements for semester project documentation

- Your semester project will be submitted as zipped archive **Surname_FirstName.zip** of three files.
- The first file named **Surname_FirstName_CPU.v** should contain all source codes in Verilog.

Include the Verilog description only for the CPU. Do not include the description of other components (data memory, instruction memory, etc.).

Use the following template:

```
`default_nettype none
module processor( input         clk, reset,
                  output [31:0] PC,
                  input  [31:0] instruction,
                  output        WE,
                  output [31:0] address_to_mem,
                  output [31:0] data_to_mem,
                  input  [31:0] data_from_mem
                );
    //... write your code here ...
endmodule

//... add new Verilog modules here ...
`default_nettype wire
```

You can add and use new Verilog modules as you wish.

- The second file named **Surname_FirstName_prog1.asm** should contain program S in the RISC-V assembly language.
- The third file named **Surname_FirstName_prog1.hex** should contain program S in the hexadecimal format (one instruction per line).

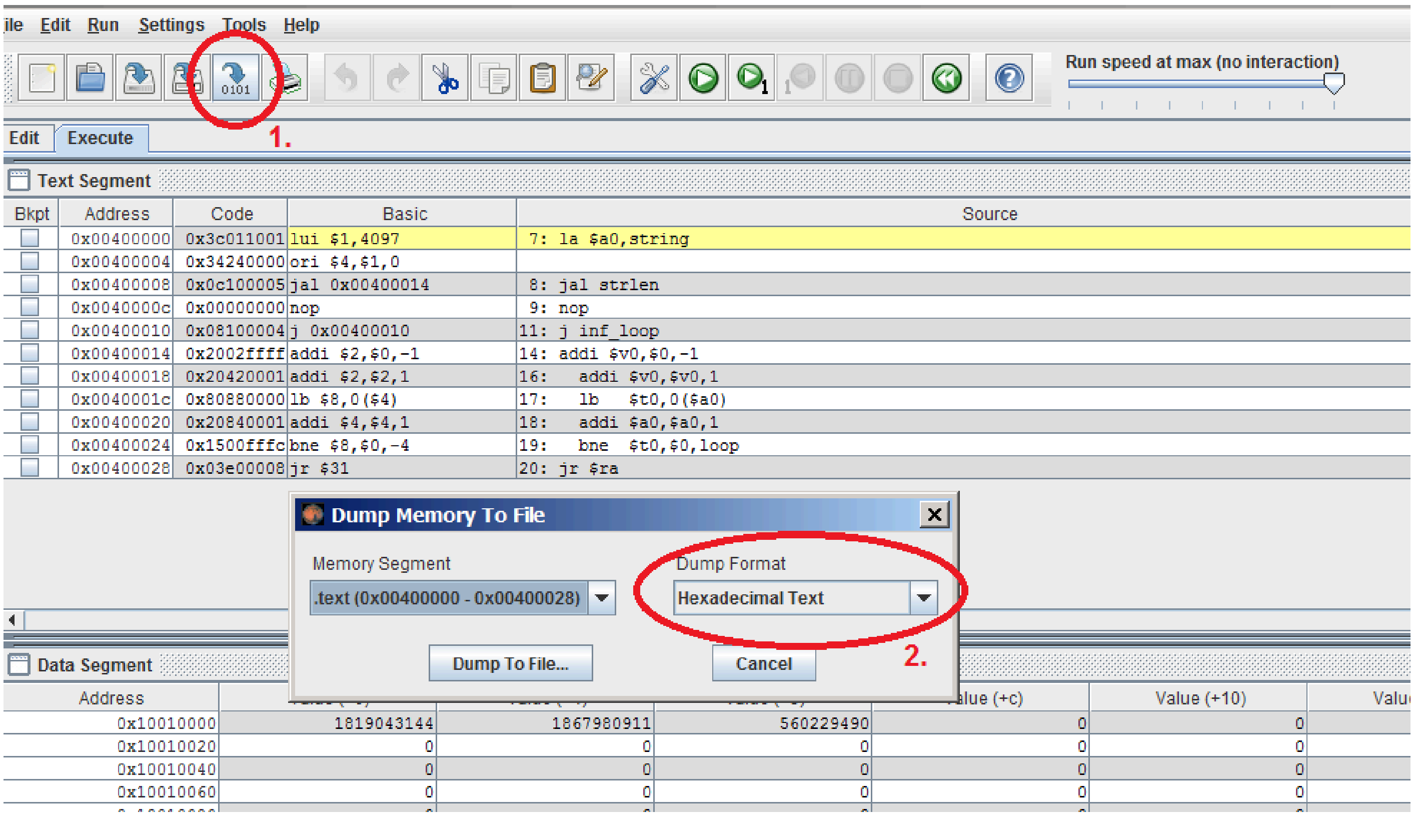
Submition of your semester project

Submit the zip archive **Surname_FirstName.zip** to the web page http://biaps.fit.cvut.cz/first_semestral_project/index.php.

You must authenticate using the last 3 digits of your ID number = the number found on your ISIC card or found on https://usermap.cvut.cz.

Hints

You can use the RARS simulator to generate the machine code of program S. See figure bellow. Note: RARS implements RISC-V ISA, which slightly differs from ISA described above (there is no addu.qb instruction).



Bkpt	Address	Code	Basic	Source
	0x00400000	0x3c011001	lui $1,4097	7: la $a0,string
	0x00400004	0x34240000	ori $4,$1,0	
	0x00400008	0x0c100005	jal 0x00400014	8: jal strlen
	0x0040000c	0x00000000	nop	9: nop
	0x00400010	0x08100004	j 0x00400010	11: j inf_loop
	0x00400014	0x2002ffff	addi $2,$0,-1	14: addi $v0,$0,-1
	0x00400018	0x20420001	addi $2,$2,1	16: addi $v0,$v0,1
	0x0040001c	0x80880000	lb $8,0($4)	17: lb $t0,0($a0)
	0x00400020	0x20840001	addi $4,$4,1	18: addi $a0,$a0,1
	0x00400024	0x1500fffc	bne $8,$0,-4	19: bne $t0,$0,loop
	0x00400028	0x03e00008	jr $31	20: jr $ra

Memory Segment: .text (0x00400000 - 0x00400028)
Dump Format: Hexadecimal Text

Address				alue (+c)	Value (+10)	Valu
0x10010000	1819043144	1867980911	560229490	0	0	
0x10010020	0	0	0	0	0	
0x10010040	0	0	0	0	0	
0x10010060	0	0	0	0	0	

You can use the following Verilog modules to represent the whole computer system. If the data and instruction memory arrays of vectors are not large enough, extend them. However, please **do not** include them into the **Surname_FirstName_CPU.v** file.

```
module top (     input          clk, reset,
                 output [31:0] data_to_mem, address_to_mem,
                 output         write_enable);

        wire [31:0] pc, instruction, data_from_mem;

        inst_mem  imem(pc[7:2], instruction);
        data_mem  dmem(clk, write_enable, address_to_mem, data_to_mem, data_from_mem);
        processor CPU(clk, reset, pc, instruction, write_enable, address_to_mem, data_to_mem, da
endmodule

//-------------------------------------------------------------------
module data_mem (input clk, we,
                 input  [31:0] address, wd,
                 output [31:0] rd);

        reg [31:0] RAM[63:0];

        initial begin
                $readmemh ("memfile_data.hex",RAM,0,63);
        end

        assign rd=RAM[address[31:2]]; // word aligned
        always @ (posedge clk)
                if (we)
                        RAM[address[31:2]]<=wd;
endmodule

//-------------------------------------------------------------------
module inst_mem (input  [5:0]  address,
                 output [31:0] rd);

        reg [31:0] RAM[63:0];
        initial begin
                $readmemh ("memfile_inst.hex",RAM,0,63);
        end
        assign rd=RAM[address]; // word aligned
endmodule
```

And for the simulation, you can use the following template:

```
module testbench();
        reg         clk;
        reg         reset;
        wire [31:0] data_to_mem, address_to_mem;
        wire        write_enable;

        top simulated_system (clk, reset, data_to_mem, address_to_mem, write_enable);

        initial begin
                $dumpfile("test");
                $dumpvars;
                reset<=1; # 2; reset<=0;
                #100;
                $writememh ("memfile_data_after_simulation.hex",simulated_system.dmem.RAM,0,63);
                $finish;
        end

        // generate clock
        always  begin
                clk<=1; # 1; clk<=0; # 1;
        end
endmodule
```