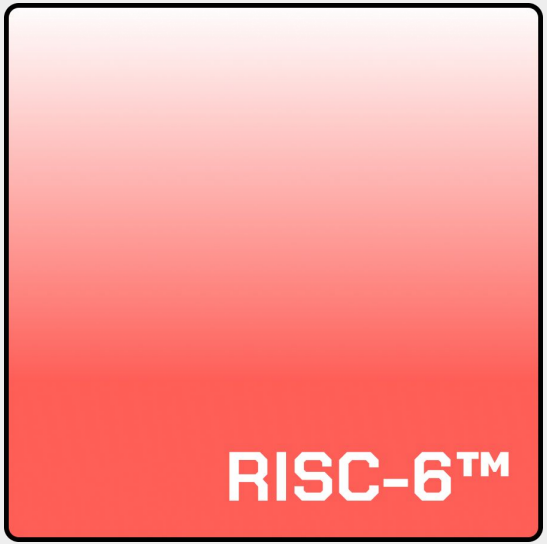
MC851 Projeto em Computação

Entrega 4

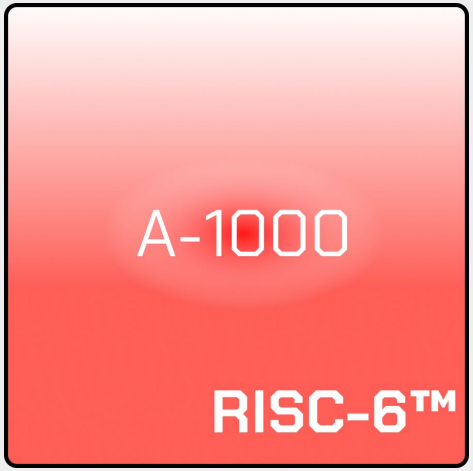
Equipe "RISC-VI":
RA 169374, Daniel Paulo Garcia
RA 182783, Lucca Costa Piccolotto Jordão
RA 185447, Paulo Barreira Pacitti
RA 198435, Guilherme Tavares Shimamoto
RA 216116, Gabriel Braga Proença
RA 221859, Mariana Megumi Izumizawa



Available now

. 5-stage pipeline RV32I* CPU Core;
. Buttons and LEDs;

Coming this Christmas



. 5-stage pipeline RV32I CPU Core;
. Button, UART and LEDs;
. Caches L1 with unified MMU;
. Faster CPU;

CPU de 5 estágios

- CPU de 5 estágios capaz de executar instruções aritméticas/lógicas do **tipo I e R**, instruções de **load** e **store**, além de resolver **branches** e **jumps**.
- CPU também resolve a maioria dos hazards de dados, exceto o "Use after load".

IF: Busca

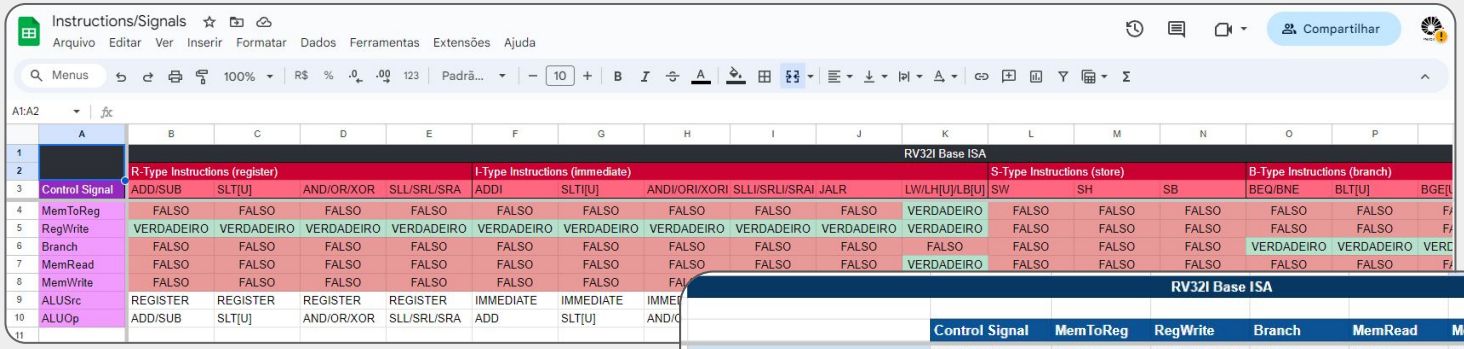
ID: Decodificação e leitura de registradores

EX: Execução

MEM: Acesso à memória

WB: Escrita de resultados

Planilha com sinais



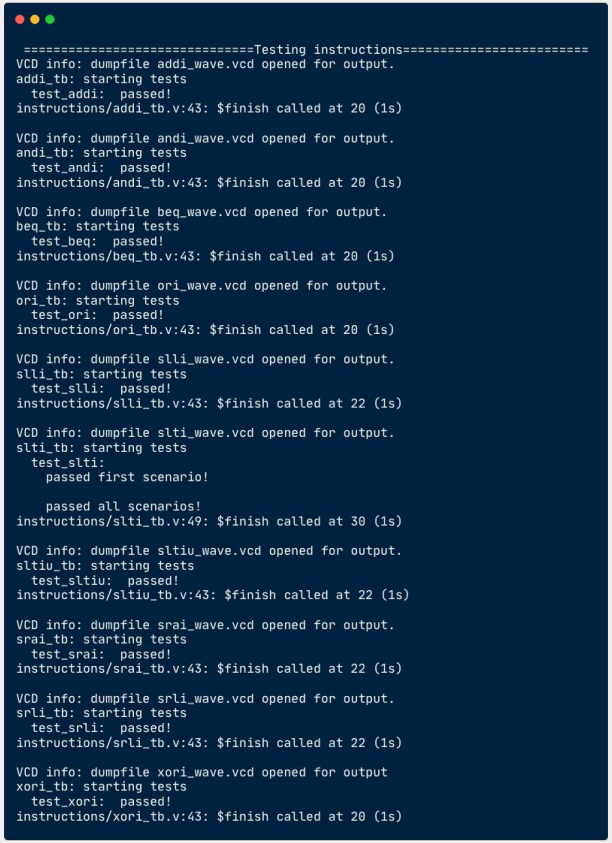
Matriz das instruções

RV32I Base ISA

	Control Signal	MemToReg	RegWrite	Branch	MemRead	MemWrite	ALUSrc	ALUOp
R-Type Instructions (register)	ADD/SUB	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	REGISTER	ADD/SUB
	SLT[U]	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	REGISTER	SLT[U]
	AND/OR/XOR	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	REGISTER	AND/OR/XOR
	SLL/SRL/SRA	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	REGISTER	SLL/SRL/SRA
I-Type Instructions (immediate)	ADDI	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	IMMEDIATE	ADD
	SLTI[U]	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	IMMEDIATE	SLT[U]
	ANDI/ORI/XORI	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	IMMEDIATE	AND/OR/XOR
	SLLI/SRLI/SRAI	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	IMMEDIATE	SLL/SRL/SRA
	JALR	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	IMMEDIATE	ADD*
	LW/LH[U]/LB[U]	VERDADEIRO	VERDADEIRO	FALSO	VERDADEIRO	FALSO	IMMEDIATE	ADD
S-Type Instructions (store)	SW	FALSO	FALSO	FALSO	FALSO	VERDADEIRO	IMMEDIATE	ADD
	SH	FALSO	FALSO	FALSO	FALSO	VERDADEIRO	IMMEDIATE	ADD
	SB	FALSO	FALSO	FALSO	FALSO	VERDADEIRO	IMMEDIATE	ADD
B-Type Instructions (branch)	BEQ/BNE	FALSO	FALSO	VERDADEIRO	FALSO	FALSO	REGISTER	SUB
	BLT[U]	FALSO	FALSO	VERDADEIRO	FALSO	FALSO	REGISTER	SUB
	BGE[U]	FALSO	FALSO	VERDADEIRO	FALSO	FALSO	REGISTER	SUB
U-Type Instructions (upper-immediate)	LUI	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	IMMEDIATE	ADD
	AUIPC	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	IMMEDIATE	ADD
J-Type Instructions (jump)	JAL	FALSO	VERDADEIRO	FALSO	FALSO	FALSO	IMMEDIATE	ADD

Instruções

- R-Immediate:
  - addi
  - andi
  - ori
  - slli
  - slti
  - sltiu
  - srai
  - srli
  - xori
- R-Register:
  - add
  - and
  - or
  - sll
  - slt
  - sltu
  - sra
  - srl
  - xor
- Control transfer:
  - jal
  - jalr
  - beq
  - bge
  - bgeu
  - blt
  - bltu
  - bne
- Load/Store (A-500*):
  - lw
  - sw



Saída dos testes automatizados *clock-accurate* das instruções da ISA

SoC: *system-on-a-chip*

- *top module*;
- Conecta a CPU com a memória. Módulo central do sistema computacional implementado utilizando a arquitetura RISC-V
- A-500: MMU does not exist, memory and peripherals are linked individually in the CPU

```
module soc #(
    parameter ROMFILE="../src/memdump/addi.mem"
) (
    input clk,
    input btn1,
    input btn2,
    output [5:0] led,
    output uart_tx
);
    wire         mmu_mem_ready;
    wire [31:0]  mmu_data_out;
    wire         mmu_write_enable;
    wire         mmu_read_enable;
    wire         mmu_mem_signed_read;
    wire         mmu_signed_read;
    wire [ 1:0]  mmu_mem_data_width;
    wire [31:0]  mmu_address;
    wire [31:0]  mmu_data_in;

    cpu cpu_inst (
        .clk (clk),
        .reset_n (btn2),
        .mmu_mem_ready(mmu_mem_ready),
        .mmu_data_out(mmu_data_out),
        .mmu_write_enable(mmu_write_enable),
        .mmu_read_enable(mmu_read_enable),
        .mmu_mem_signed_read(mmu_signed_read),
        .mmu_mem_data_width(mmu_mem_data_width),
        .mmu_address(mmu_address),
        .mmu_data_in(mmu_data_in),
        .uart_data(data)
    );

    mmu #( .ROMFILE(ROMFILE)) mmu_inst (
        .clk(clk),
        .btn1(btn1),
        .btn2(btn2),
        .reset_n(btn2),
        .write_enable(mmu_write_enable),
        .read_enable(mmu_read_enable),
        .mem_signed_read(mmu_signed_read),
        .mem_data_width(mmu_mem_data_width),
        .address(mmu_address),
        .data_in(mmu_data_in),
        .data_out(mmu_data_out),
        .led(led),
        .mem_ready(mmu_mem_ready)
    );
```

Implementação do `soc.v`, mostrando a integração entre MMU e CPU. Além dos periféricos, botões e LEDs.

Cache L1

Cache L1 de 512 linhas× 32bit (= 2KiB) mapeada diretamente, do tipo write-back com política write-allocate.

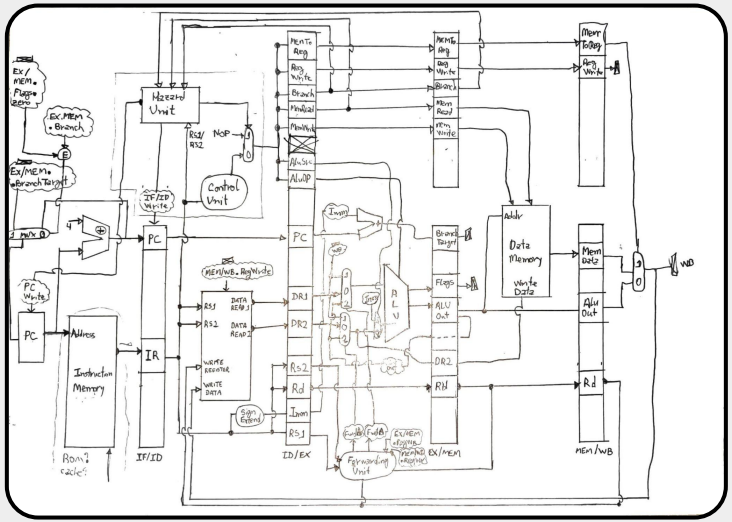
Módulo Cache L1

```
module l1_cache #(
    parameter INDEX_WIDTH = 6
) (
    input  wire         clk,
    input  wire         reset_n,
    input  wire         write_enable,
    input  wire         read_enable,
    input  wire [31:0]  address,
    input  wire [31:0]  data_in,
    output wire [31:0]  data_out,
    output wire         cache_ready,

    // Memory
    input  wire         mem_ready,
    output wire         mem_fetch,
    output wire         mem_write
);
    localparam OFFSET_WIDTH     = 2;
    localparam TAG_WIDTH        = 32 - (INDEX_WIDTH + OFFSET_WIDTH);
    localparam NUM_OF_BLOCKS    = 2**INDEX_WIDTH;

    // TODO: Implementar memória de cache com duas B-SRAMs Single-Port (flags+tag e dados)
    reg block_valid [0:NUM_OF_BLOCKS-1];
    reg block_dirty [0:NUM_OF_BLOCKS-1];
    reg [TAG_WIDTH-1:0] tag_array [0:NUM_OF_BLOCKS-1];
    reg [31:0] block_data [0:NUM_OF_BLOCKS-1];

    initial begin: label0
        integer i;
        for (i = 0; i < NUM_OF_BLOCKS; i = i + 1) begin
            block_valid[i] = 0;
            block_dirty[i] = 0;
            tag_array[i] = 0;
            block_data[i] = 0;
        end
    end

    wire [TAG_WIDTH-1:0]    tag;
    wire [INDEX_WIDTH-1:0]  index;
    wire [OFFSET_WIDTH-1:0] offset;
    wire cache_hit;

    assign tag       = address[31:INDEX_WIDTH+OFFSET_WIDTH];
    assign index     = address[INDEX_WIDTH+OFFSET_WIDTH-1 : OFFSET_WIDTH];
    assign offset    = address[OFFSET_WIDTH-1:0];

    // Cache Controller State Machine
    localparam ACCEPT_REQUEST   = 2'd0;
    localparam WRITE_BACK       = 2'd1;
    localparam MEM_ALLOCATE     = 2'd2;

    reg [1:0] ctrl_state = ACCEPT_REQUEST;

    assign cache_hit = block_valid[index] && tag_array[index] == tag;
    assign data_out = read_enable ? block_data[index] : 0;
```

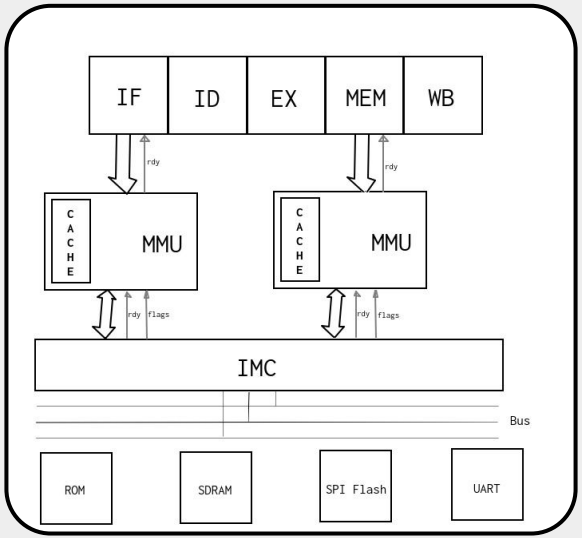
```
    always @(posedge clk, negedge reset_n) begin
        if (!reset_n) begin
            ctrl_state = ACCEPT_REQUEST;
        end else begin
            case (ctrl_state)
                ACCEPT_REQUEST: begin
                    if (!cache_hit && block_dirty[index]) begin
                        ctrl_state <= WRITE_BACK;
                    end else if (!cache_hit && !block_dirty[index]) begin
                        ctrl_state <= MEM_ALLOCATE;
                    end else begin
                        block_valid[index] <= 1;
                        tag_array[index] <= tag;

                        if (write_enable) begin
                            block_dirty[index] <= 1;
                            block_data[index] <= data_in;
                        end
                    end
                end

                WRITE_BACK: begin
                    if (mem_ready) begin
                        ctrl_state <= MEM_ALLOCATE;
                    end
                end

                MEM_ALLOCATE: begin
                    if (mem_ready) begin
                        block_valid[index] <= 1;
                        block_dirty[index] <= 0;
                        tag_array[index] <= tag;
                        block_data[index] <= data_in;

                        ctrl_state <= ACCEPT_REQUEST;
                    end
                end

                default: begin
                    ctrl_state <= ACCEPT_REQUEST;
                end
            endcase
        end
    end

    always @(*) begin
        cache_ready = 0;
        mem_fetch = 0;
        mem_write = 0;

        case (ctrl_state)
            ACCEPT_REQUEST: begin
                cache_ready = 1;
            end

            WRITE_BACK: begin
                mem_write = 1;
            end

            MEM_ALLOCATE: begin
                mem_fetch = 1;
            end

            default: ;
        endcase
    end
endmodule
```

MMU

Memory Control: **controle de priorização de acesso à memória** por diferentes estágios da pipeline e resolução de conflitos de acesso.



Danielpath: datapath utilizado no sistema computacional.

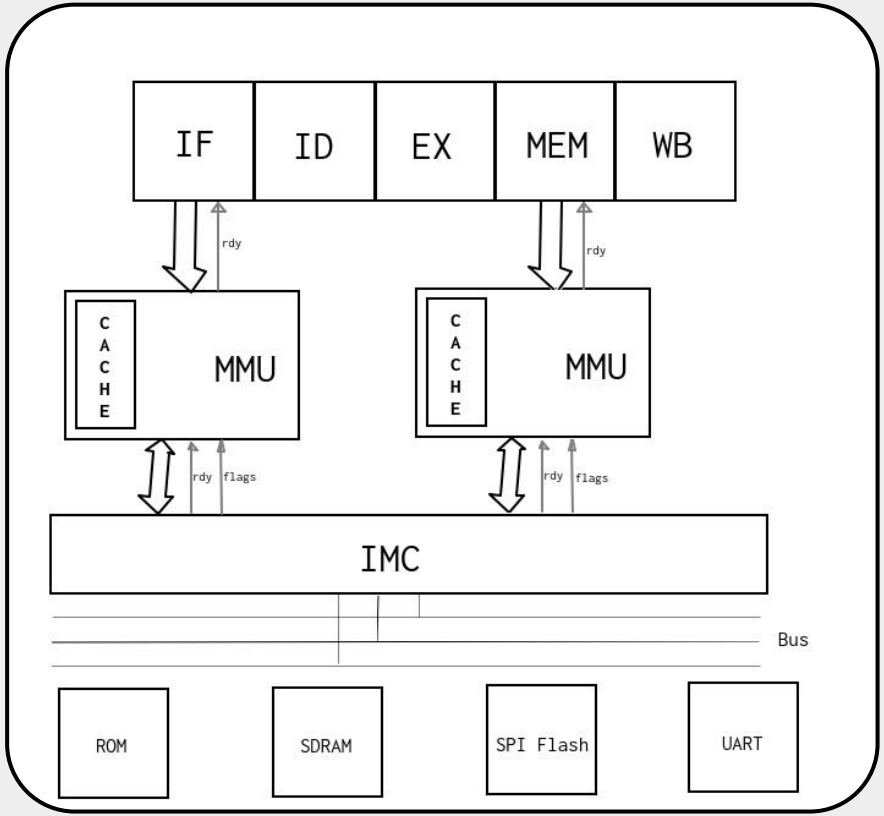


Danielpath: memória no A-1000

MMU

O que faltou implementar:

- Barramento de memória unificado, que deveria mapear endereços físicos pros respectivos dispositivos;
- Responder à MMU quais a flags de permissão de acesso para cada região de memória (necessário para MMIO e periféricos funcionar);
- Modificar a máquina de estados da MMU para funcionar com essa arquitetura;
- Passar o controle de acesso pro IMC.



Danielpath: memória no A-1000

Periféricos

Botões da placa FPGA podem ser acessados através da MMU, permitindo uma leitura através de LOADs.

- Módulo do botão com buffers para manutenção dos estados;
- Teste do módulo;
- Interface com a MMU;
- Teste de leitura do módulo através da MMU.

```
module btn #(
    parameter ADDR_WIDTH = 8
) (
    input clk,
    input btn1,
    input btn2,
    input read_enable,
    input [ADDR_WIDTH-1:0] address,
    output reg [31:0] data_out
);

    reg btn1_buffer;
    reg btn2_buffer;

    always @(posedge clk) begin
        if (!btn1)
            btn1_buffer <= ~btn1;

        if (!btn2)
            btn2_buffer <= ~btn2;

        if (read_enable) begin
            case (address)
                0: begin
                    data_out <= btn1_buffer;
                    btn1_buffer <= 0; // reset bt1_buffer after
reading, runs in the next clock cycle
                end
                1: begin
                    data_out <= btn2_buffer;
                    btn2_buffer <= 0; // reset bt2_buffer after
reading, runs in the next clock cycle
                end
                default: data_out <= 0;
            endcase
        end else begin
            data_out <= 0;
        end
    end
endmodule
```

Módulo do botão com buffers para manutenção dos estados

Periféricos

LEDs da placa FPGA podem ser acessados através da MMU, permitindo uma leitura através de LOADs.

- Módulo do LED;
- Teste do módulo;
- Interface com a MMU;
- Teste de leitura do módulo através da MMU.

```
module led #(
    parameter ADDR_WIDTH = 8
)(
    input clk,
    input write_enable,
    input [ADDR_WIDTH-1:0] address,
    input [31:0] data_in,
    output [5:0] led
);

    reg [5:0] led_data_out;

    always @(posedge clk) begin
        if (write_enable) begin
            led_data_out <= data_in[5:0];
        end

    end

    assign led = led_data_out;

endmodule
```

Módulo de LED

Periféricos

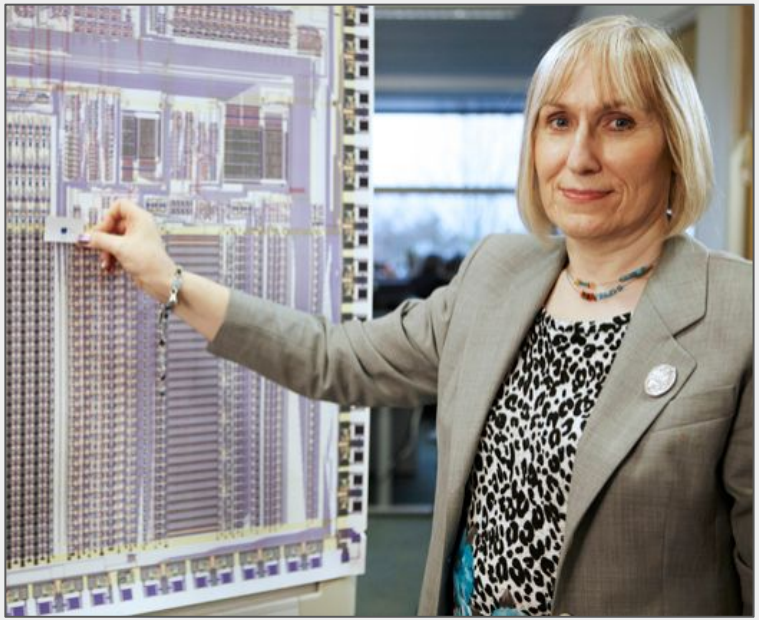
```
`include "define.v"

module mmu #(
    parameter ROMFILE="../src/memdump/addi.mem"
) (
    input clk, reset_n, btn1, btn2,
    input write_enable,
    input read_enable,
    input mem_signed_read,
    input [ 1:0] mem_data_width,
    input [31:0] address,
    input [31:0] data_in,
    output reg [31:0] data_out,
    output [5:0] led,
    output reg mem_ready
);
  ...
```

Parte da interface dos periféricos com MMU

```
...
/*****************************************************************
* BTN (Peripheral)
*/
localparam BTN_ADDR_WIDTH = 8;
reg btn_read_enable;
wire  [BTN_ADDR_WIDTH-1:0] btn_address;
wire [31:0] btn_data_out;

btn #( .ADDR_WIDTH(BTN_ADDR_WIDTH) ) btn_inst (
  .clk            (clk                ),
  .btn1           (btn1               ),
  .btn2           (btn2               ),
  .read_enable    (btn_read_enable    ),
  .address        (btn_address        ),
  .data_out       (btn_data_out       )
);
// ----------------------------------------------------------------

/*****************************************************************
* LED (Peripheral)
*/
localparam LED_ADDR_WIDTH = 8;
reg led_write_enable;
wire  [LED_ADDR_WIDTH-1:0] led_address;
wire [31:0] led_data_in;

led #( .ADDR_WIDTH(LED_ADDR_WIDTH) ) led_inst (
  .clk            (clk                ),
  .write_enable   (led_write_enable   ),
  .address        (led_address        ),
  .data_in        (led_data_in        ),
  .led            (led                )
);
// ----------------------------------------------------------------
...
```

Demonstração

"In July 1981, Wilson extended the Acorn Atom's BASIC programming language dialect into an improved version for the Acorn Proton, a microcomputer that enabled Acorn to win the contract with the British Broadcasting Corporation (BBC) for their ambitious computer education project. Hauser employed a deception, telling both Wilson and colleague Steve Furber that the other had agreed **a prototype could be built within a week.** Taking up the challenge, she designed the system including the circuit board and components from Monday to Wednesday, which required fast new DRAM integrated circuits to be sourced directly from Hitachi. **By Thursday evening**, a prototype had been built, **but the software had bugs, requiring her to stay up all night and into Friday debugging**. Wilson recalled watching the wedding of Prince Charles and Lady Diana Spencer on a small portable television while attempting to debug and re-solder the prototype. It was a success with the BBC, who awarded Acorn the contract."



Sophie Wilson near a photograph of the first ARM processor holding a photograph of the ARM Cortex-M0+ to the same scale.

Demonstração

Entrega 1 ✅

Planejar e desenvolver um processador RISC-V de 32 bits, com pipeline, que suporte o conjunto RV32I.

- Compreender e desenvolver um **pipeline** para o processador RISC-V;
- Compreender e desenvolver **programas de teste** para o processador;
- Compreender e desenvolver um **ambiente de execução** (restrito ao momento) para o **processador**;
- Compreender e utilizar o **ambiente de execução em FPGA** para implementar o **processador**;

Entrega 2

Conjunto de instruções do processador deve ser incrementado para suportar RV32IMA.

- Compreender e desenvolver a **cache L1**;
- Compreender e desenvolver um **periférico** para o processador;
- Compreender e desenvolver um **ambiente de execução** (restrito ao momento) para o **periférico**;
- Compreender e utilizar o **ambiente de execução em FPGA** para implementar o **periférico**;
- Demonstrar um **código** que utilize as **instruções** e o **periférico** em FPGA;

Entrega 3

Conjunto de instruções do processador deve ser incrementado para suportar as instruções compactas.

- Compreender e desenvolver um **periférico** para o processador;
- Compreender e desenvolver um **ambiente de execução** (restrito ao momento) para o **periférico**;
- Compreender e utilizar o **ambiente de execução em FPGA** para implementar o **periférico**;
- Demonstrar um código que utilize as **instruções** e os **periféricos** em FPGA;

Maiores desafios

Autonomia e Desafios na Disciplina

Autonomia nas Disciplinas de Projeto

Desafios Iniciais e Necessidade de Estudo: O início do projeto foi o maior desafio devido à necessidade de estudar conceitos de MC732 e superar dificuldades na criação e configuração do ambiente.

Paradigma no Desenvolvimento em Hardware: A compreensão do paradigma no desenvolvimento em hardware com Verilog foi um desafio, especialmente ao perceber que as descrições de hardware executam simultaneamente.

Datapath Completo e Integração com MMU: Compreender o funcionamento de um datapath completo de uma CPU e a integração complexa, especialmente com a MMU, gerou complicações, notadamente nas instruções de Load/Store.

Maiores desafios

Desafios na Implementação do Projeto

Detalhes Arquiteturais Complexos:

- Enfrentamos desafios significativos em detalhes arquiteturais que frequentemente são omitidos por simplicidade em livros-texto;
- A interface de memória da CPU foi especialmente desafiadora, divergindo da representação comum de uma caixa preta na literatura.

Realidade Diferente da Literatura:

- Contrariamente à visão simplificada da literatura, descobrimos que a interface de memória não realiza leituras e escritas simultâneas de forma transparente;
- Casos em que um recurso de memória está ocupado exigem a interrupção da pipeline, destacando a complexidade real dessas operações.

O que faríamos de diferente?

Se fosse possível fazer o projeto novamente

Entregas Incrementais: Adotamos uma abordagem de entregas mais simples e progressivas, implementando funcionalidades de forma gradual.

Design Multiciclo em Vez de Pipeline: Optamos por uma CPU mais simples, com um design multiciclo em vez de uma pipeline verdadeira, priorizando implementações mais simples nas primeiras entregas.

Uso Intensivo de Máquinas de Estado: Utilizamos extensivamente máquinas de estado para operações complicadas, mesmo que isso demandasse mais ciclos, visando a entrega de mais recursos ao longo do desenvolvimento.

Transição para Ambiente da Gowin: Inicialmente, dedicamos mais tempo à execução do código na FPGA, migrando do Yosys para o ambiente da Gowin. A escolha foi motivada pela complexidade do projeto, para o qual o Yosys não apresentou uma síntese suficiente.

O que faríamos de diferente?

Se fosse possível fazer o projeto novamente

Foco na Resolução de Desafios em Memória: Investimos considerável tempo na resolução de desafios relacionados à memória, reconhecendo-a como a área de maior dificuldade e número significativo de refatorações.

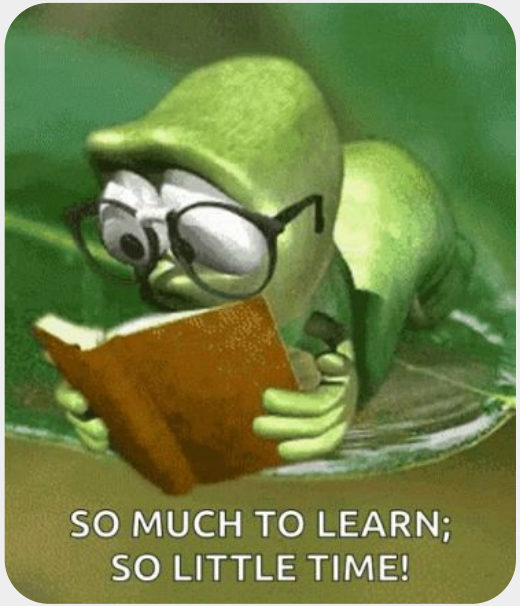
Dedicação do Grupo: Apesar das entregas não terem sido completas, o grupo demonstrou dedicação com extensas horas de estudo e colaboração. Implementamos um sistema computacional consistente, destacando a importância de code reviews e testes automatizados no processo.

Maiores aprendizados

- Utilização do **Verilog** 2005 e suas armadilhas;
- **Compreensão** da stack de software necessária para desenvolver, testar e debugar sistemas em **FPGA**, diferenciando entre código HDL sintetizável e simulação;
- Prática na **concepção e implementação do datapath de uma CPU RISC-V com pipeline**;
- Pesquisa de informações técnicas sobre a **FPGA Gowin** e a **placa Tang Nano 9K**;
- Desenvolvimento de habilidades para **trabalhar em equipe**, promovendo contribuições mútuas e aprendizado entre os colegas.

Maiores aprendizados

- Revisão e aplicação prática de **conceitos de diversas disciplinas do curso**;
- Abordagem de **soluções mais simples para problemas complexos**, progredindo gradualmente;
- Reconhecimento da importância de focar não apenas na totalidade do processador nas primeiras entregas, mas em **construir passo a passo**;
- A maior lição foi **compreender o design e implementação de sistemas computacionais**, semelhante à realidade da produção de chips.

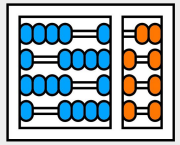


Roteiro

- O que precisamos fazer/O que fizemos
- Desafios
- Feedbacks
- O que faríamos de diferente?
- Aprendizados

Referências

1. D. A. Patterson and J. L. Hennessy, *Computer Organization and Design RISC-V Edition: The Hardware Software Interface*, 1st ed. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2017
2. Repositório com código do sitema computacional: https://github.com/izumizawa/mc851



Obrigado!

Equipe "RISC-VI":
RA 169374, Daniel Paulo Garcia
RA 182783, Lucca Costa Piccolotto Jordão
RA 185447, Paulo Barreira Pacitti
RA 198435, Guilherme Tavares Shimamoto
RA 216116, Gabriel Braga Proença
RA 221859, Mariana Megumi Izumizawa