Universidade de Brasília
Departamento de Ciência da Computação

Laboratório 2 - CPU RISC-V UNICICLO
Organização e Arquitetura de Computadores

Evelyn Soares Pereira 170102785
André Rodrigues Modesto 211068234
Théo Henrique Gallo 170080781
Eduardo de Souza Costa Assunção 211068270
Alexandre Junqueira Correia Lima 211068225

Brasília, 8 de Junho de 2025

1) Implemente o processador Uniciclo com ISA Reduzida com as instruções: add, sub, and, or, slt, lw, sw, beq, jal, e ainda as instruções jalr e addi.

1.1 - Para validar a implementação do processador, foi escrito um programa em assembly RISC-V “de1.s”, com o objetivo de testar as 11 instruções. Utilizando os registradores t1 e t2 como auxiliares e o registrador t0 armazenando resultados para verificar cada teste e facilitar a visualização dos resultados. As instruções foram testadas com valores conhecidos, e para as instruções lw e sw o programa manipula os dados utilizando o registrador sp, apontando para endereços válidos dentro da região de dados da memória.

Em .data, é declarado o rótulo valor que armazena a palavra. O programa entra no rótulo LOOP, onde inicializa t0 com 2 e t1 com 5, depois realiza operações aritméticas e lógicas entre eles, atualizando t0. A instrução la t1, valor carrega o endereço do dado valor em t1. Em seguida, lw t2, 0(t1) carrega o valor 5 (armazenado em valor) em t2, e sw t2, 4(t1) salva esse mesmo valor no endereço valor + 4. Depois, beq t0, t1, FIM verifica se t0 é igual a t1, mas como não são, não pula. A instrução jal ra, FUNCAO pula para FUNCAO e salva o endereço de retorno em ra. Dentro de FUNCAO, t0 recebe 3, e jalr zero, 0(ra) pula de novamente para VOLTA, usando o endereço salvo em ra. Em VOLTA, jal zero, FIM salta para o fim do programa que entra num loop infinito com jal zero, FIM.

1.2 - O módulo Registers foi criado para permitir três leituras simultâneas: uma para o operando rs1 que é necessária para praticamente todas as instruções que operam sobre registradores, como add e lw, outra para o operando rs2 que é necessária em instruções do tipo R e tipo S, onde dois registradores-fonte são usados, e outra adicional para visualização “disp” servindo para fins de debug e visualização. O módulo Registers também suporta uma escrita por ciclo de clock. Os registradores sp e gp são inicializados com os valores 0x1001_03FC e 0x1001_0000, respectivamente, conforme especificado.

Os índices dos registradores que vão ser lidos simultaneamente e vêm diretamente dos campos rs1 e rs2 da instrução são:

```
input wire  [4:0]       iReadRegister1, iReadRegister2;
```

As duas saídas de leitura simultânea que conectam o banco de registradores à partes do processador são:

```
output wire [31:0]      oReadData1, oReadData2;
```

Já as três instruções que fazem três leituras simultâneas do banco de registradores, acessando três posições diferentes (ou iguais, dependendo da instrução) são: oReadData1 recebe o valor do registrador apontado por iReadRegister1, oReadData2 recebe o valor do registrador apontado por iReadRegister, oRegDisp recebe o valor do registrador que será mostrado.

No bloco always @(posedge iCLK or posedge iRST), se iRegWrite estiver ativo e o registrador destino não for x0, o valor é escrito:

```
if(iRegWrite && (iWriteRegister != 5'b0))
   registers[iWriteRegister] <= iWriteData;
```

x0 é sempre zero, então não pode ser escrito.

1.3 - O módulo ImmGen é responsável por extrair e estender corretamente os campos imediatos das instruções, conforme o tipo de instrução. Este módulo é responsável por interpretar os bits da instrução RISC-V e extrair o campo de imediato, que pode estar distribuído em partes diferentes do binário dependendo do tipo de instrução. Com a extensão de sinal implementada para garantir que os valores negativos sejam representados corretamente.

No trecho:
always @ (*)
  case (iInstrucao[6:0])

A parte [6:0] é o opcode da instrução, que define o tipo dela (load, store, branch, etc.). Com base nisso, é escolhido como montar o imediato.
oImm <= {{20{iInstrucao[31]}}, iInstrucao[31:20]};
Pega o campo [31:20] e faz sinal de extensão, repete o bit de sinal 20 vezes para formar um número com sinal.

Usado para loads (lw), operações imediatas (addi, etc.) e saltos relativos (jalr).
oImm <= {{20{iInstrucao[31]}}, iInstrucao[31:25], iInstrucao[11:7]};
Combina partes de dois campos da instrução: [31:25] e [11:7]
Junta e estende o sinal, formando um número com sinal de 12 bits.

Usado para instruções de store (sw, etc.)
oImm <= {{20{iInstrucao[31]}}, iInstrucao[7], iInstrucao[30:25], iInstrucao[11:8], 1'b0};
Juntamos tudo na ordem correta e colocamos 1'b0 no final porque o desvio é múltiplo de 2 (offset). Também faz sinal de extensão.

Usado para instruções de desvio condicional (beq)
oImm <= {{12{iInstrucao[31]}}, iInstrucao[19:12], iInstrucao[20], iInstrucao[30:21], 1'b0};

Também com um zero no fim porque o endereço é múltiplo de 2.

Usado para jal (salto com link).
oImm <= ZERO;
Se o opcode não for reconhecido, o imediato é 0.

O uso de {{N{bit}}} é sinal de extensão: copia o bit de sinal, bit mais à esquerda, para completar o valor até 32 bits. O formato final sempre é um número de 32 bits com sinal, mesmo que o imediato original tenha apenas 12, 20, etc. O resultado (oImm) será usado em operações de carga (lw), soma com imediato (addi), saltos (jal), etc.

1.4 - Após compilar o programa “de1.s”, acessamos a opção File > Dump Memory, selecionando o formato MIF 32 e exportamos o conteúdo das memórias para um arquivo de1 (sem extensão). Com isso, foram automaticamente gerados dois arquivos:

de1_text.mif – contém as instruções compiladas, iniciando no endereço 0x0040_0000;
de1_data.mif – contém os dados, começando no endereço 0x1001_0000.

Esses arquivos foram utilizados para inicializar as memórias no projeto do processador no FPGA.

1.5 - ULA mínima necessária:

Uma ULA mínima necessária implica que operações mais complexas e que exigiriam mais recursos de hardware, como multiplicação, divisão, deslocamentos complexos ou operações com números sem sinal, além das especificadas, foram intencionalmente omitidas nesta fase do projeto.

A primeira versão da ALU já apresentada no arcabouço do projeto é mais abrangente, visando suportar um conjunto mais amplo de operações, já a ULA implementada foi desenvolvida para atender às exigências da ISA RV32I reduzida, suportando as operações fundamentais requeridas pelas instruções add, sub, and, or, slt, além de prover um sinal de comparação para instruções de desvio (beq).

O módulo da ULA recebe dois operandos de 32 bits (a e b) e um código de operação de 4 bits (op). De acordo com esse código, a ULA executa uma das seguintes operações:

AND lógico bit a bit (op = 0000), OR lógico bit a bit (op = 0001), Soma de inteiros com sinal (op = 0010), Subtração (op = 0110), Set Less Than (SLT): retorna 1 se a < b (op = 0111)

Além disso, a ULA calcula um sinal zero que é ativado quando o resultado da operação é igual a zero. Esse sinal é utilizado pelo bloco de controle da CPU para determinar se um desvio condicional (beq) deve ser tomado.

1.6 - Bloco controlador e controlador da ULA

Nesta etapa do projeto, foi implementado o bloco controlador principal responsável por gerar todos os sinais de controle necessários para coordenar a execução de instruções no processador uniciclo. A arquitetura do processador baseia-se na ISA RV32I reduzida, e o controlador precisa interpretar corretamente o campo opcode da instrução para ativar ou desativar sinais como RegWrite, ALUSrc, MemRead, MemWrite, Branch, entre outros.

O módulo ControlUnit recebe como entrada o campo opcode da instrução e produz como saídas os sinais de controle. A lógica do módulo segue um modelo always_comb com um case sobre o opcode, em que cada caso define os sinais necessários para a execução correta da instrução correspondente. Para segurança e clareza, todos os sinais são inicializados com valores padrão no início do bloco.

O sinal ALUOp é um código auxiliar que é interpretado posteriormente por um segundo módulo, chamado ALUControl, que recebe também os campos funct3 e funct7 da instrução. Isso permite que a operação exata da ALU seja determinada de forma flexível, mesmo que o opcode sozinho não seja suficiente.

Para complementar o bloco controlador, foi desenvolvido o módulo ALUControl, que interpreta os sinais ALUOp, funct3 e funct7 e gera o código de operação exato para a ULA. Esse módulo é essencial para o suporte a diferentes instruções tipo R e I, como add, sub, and, or, slt e addi.

O sinal ALUOp é fornecido pelo bloco controlador principal, e indica, de forma genérica, o tipo de operação. O ALUControl detalha esse comando com base nas instruções:

Para lw e sw (ALUOp = 00): a operação é sempre soma.

Para beq (ALUOp = 01): a operação é subtração, para comparação de igualdade.

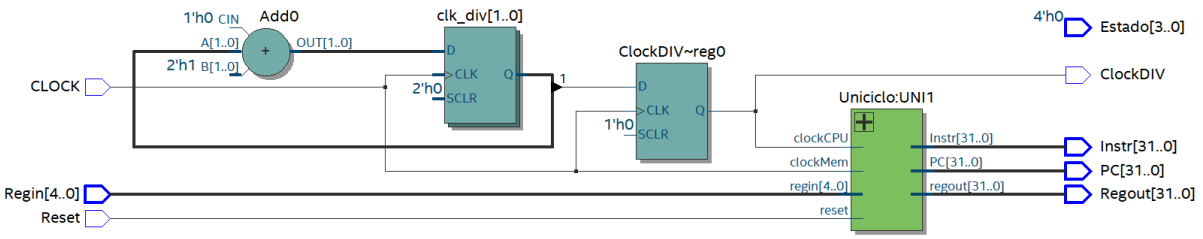
Para instruções tipo R (ALUOp = 10): o módulo examina funct3 e funct7 para diferenciar entre add e sub, além de identificar operações como and, or e slt.

Para tipo I (ALUOp = 11): examina funct3 para selecionar a operação (addi, por exemplo).

Esse controle detalhado permite a reutilização de circuitos da ULA com diferentes instruções e mantém a separação clara entre o bloco de controle geral e a lógica da ULA, tornando o projeto mais modular e escalável.

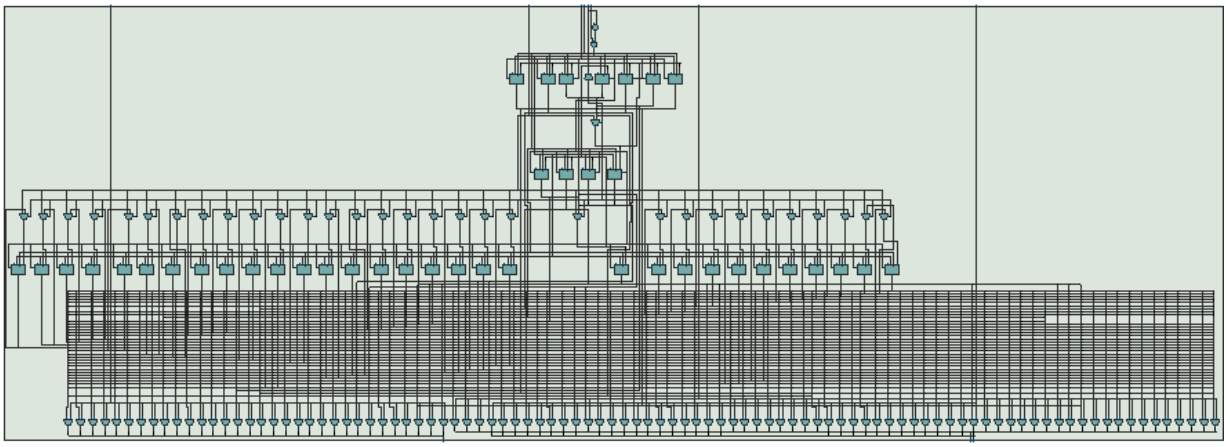
1.7 - O processador foi implementado com base no modelo uniciclo, fazendo a codificação em Verilog dos módulos e a interconexão correta desses módulos no módulo *top-level* do processador. Todos os módulos foram integrados e o caminho de dados completo foi validado com o programa de1.s.

a) No Quartus: em Processing -> Start -> Start Analysis & Elaboration foi transformado o nosso código Verilog em uma representação de netlist lógico. Após a análise e elaboração bem-sucedida, em Tools -> Netlist Viewers -> RTL Viewer foi exibido um diagrama de blocos do nosso design.

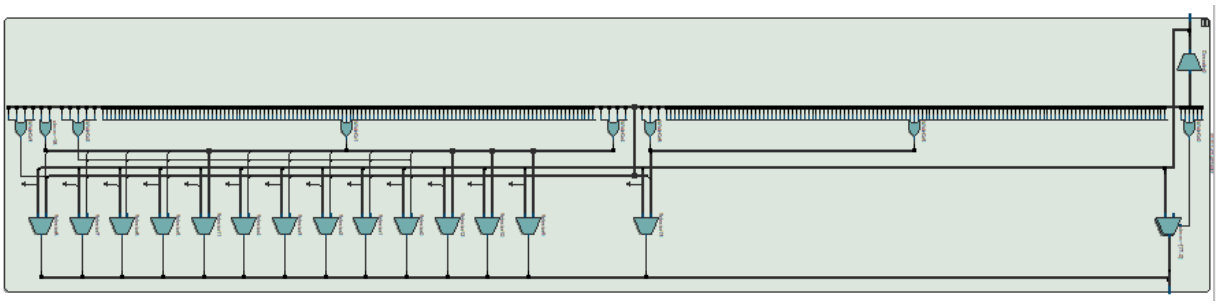


Foram gerados print screens dos principais módulos.

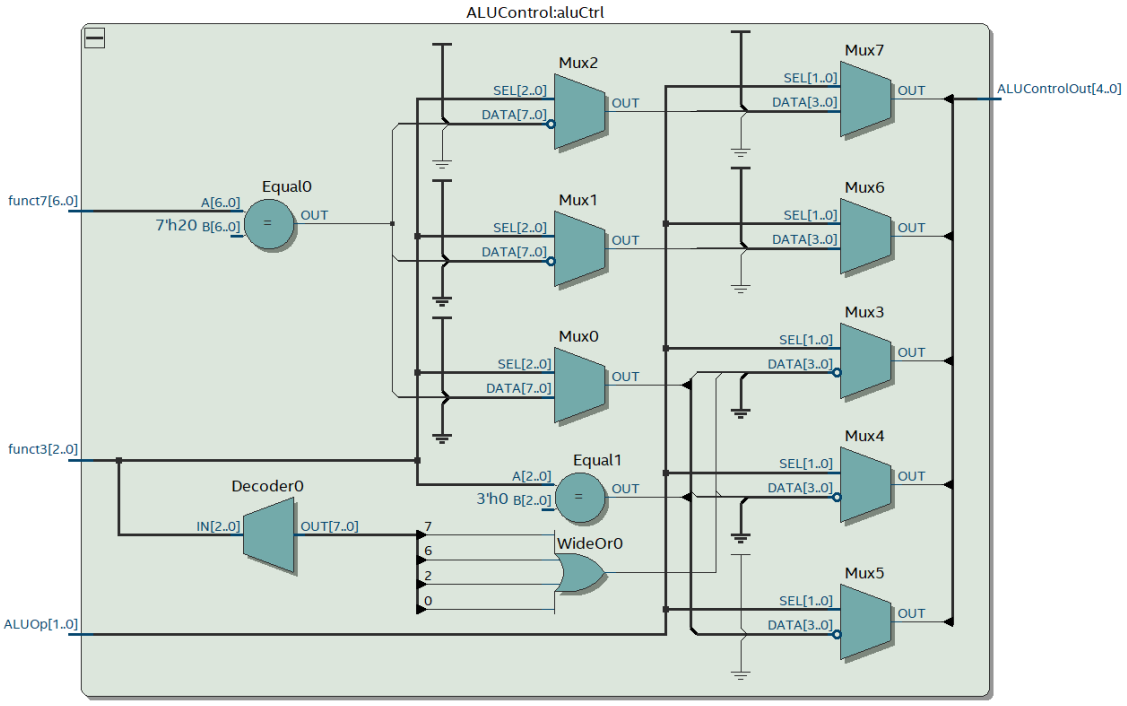
Banco de Registradores:



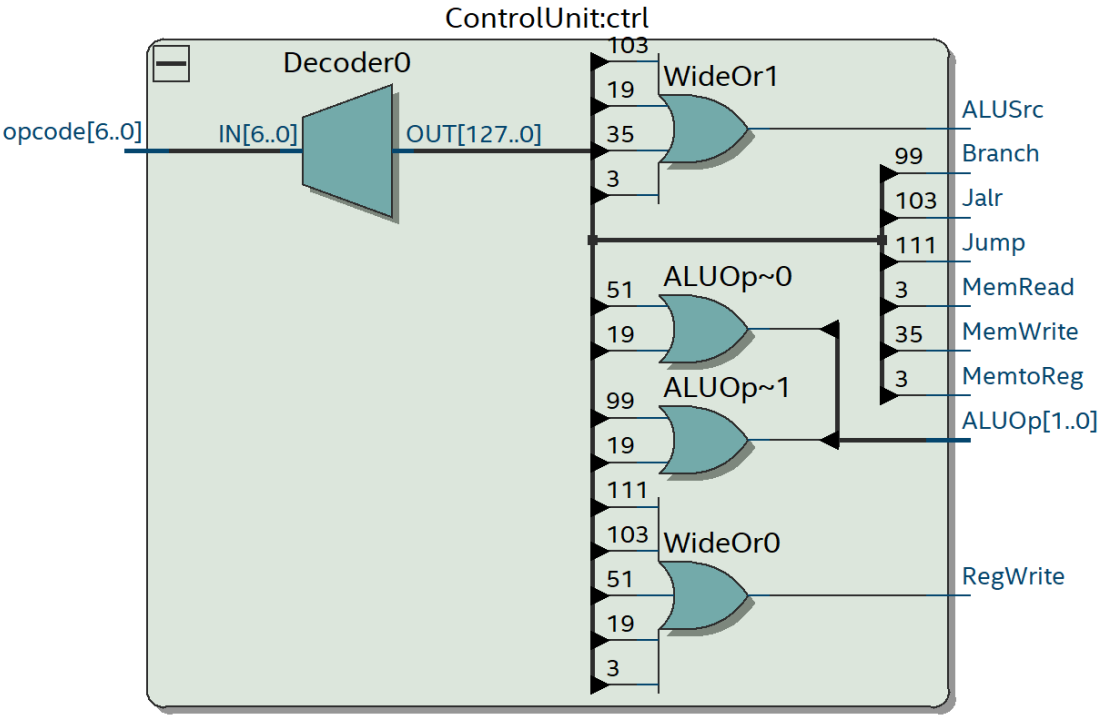
Gerador de Imediatos:



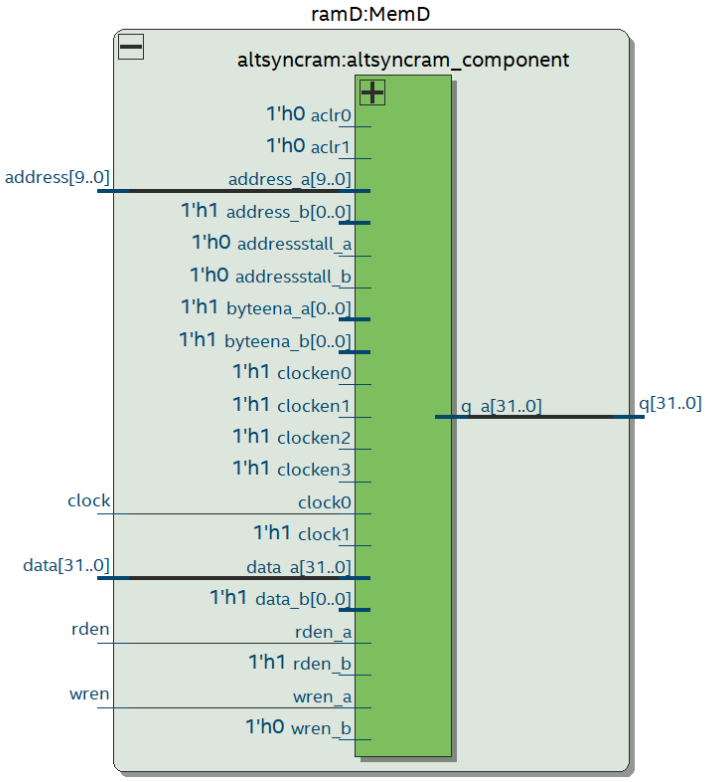
Controle da ULA:

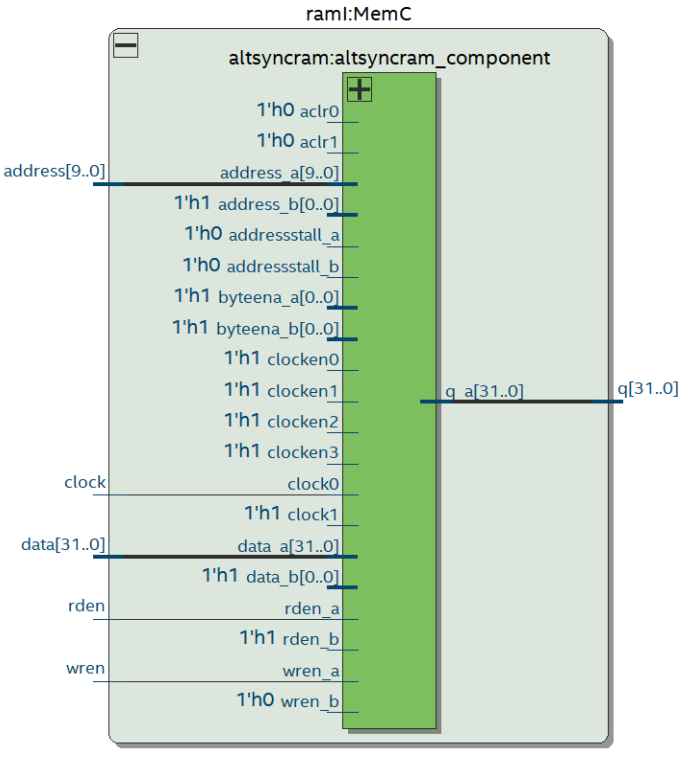


Bloco de Controle:

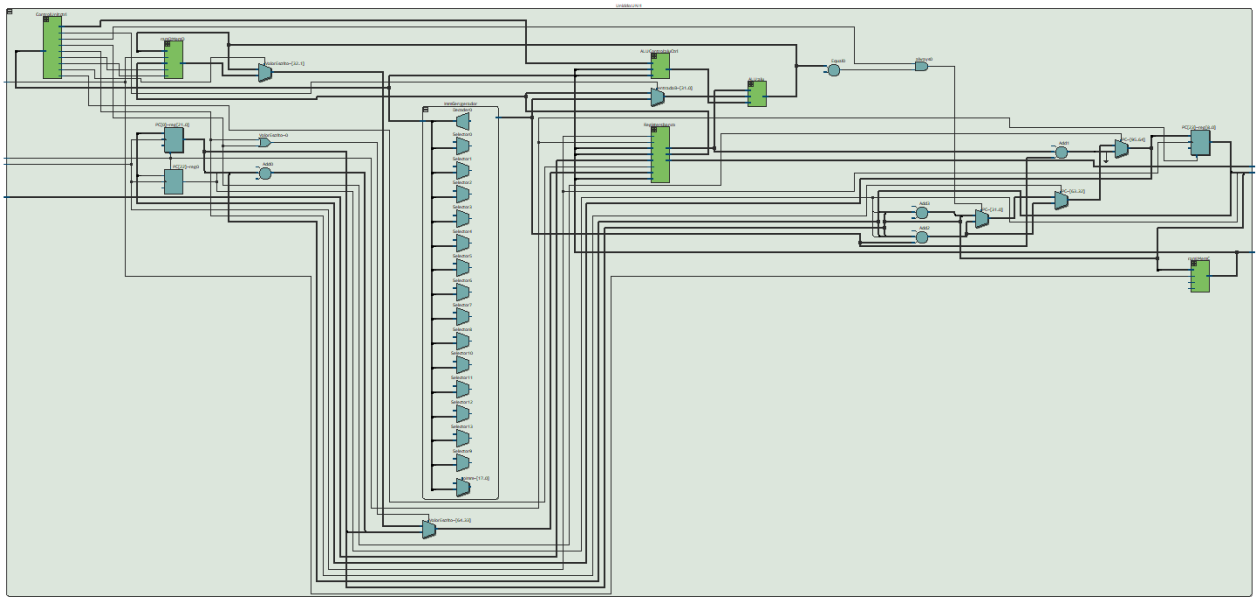


E conexões internas:





As conexões estão corretamente alinhadas com o diagrama de blocos do processador uniciclo:



b) Após a síntese do projeto, foram analisados os requisitos físicos e temporais. Em Processing -> Start -> Start Compilation, o Quartus gerou um Compilation Report. Em Project Navigator -> Flow -> Fitter -> Resource Usage Summary:

Flow Summary

<<Filter>>

Flow Status	Successful - Sun Jun 8 17:28:20 2025
Quartus Prime Version	24.1std.0 Build 1077 03/04/2025 SC Lite Edition
Revision Name	TopDE
Top-level Entity Name	TopDE
Family	Cyclone IV E
Device	EP4CE115F29C7
Timing Models	Final
Total logic elements	3,144 / 114,480 (3 %)
Total registers	1271
Total pins	108 / 529 (20 %)
Total virtual pins	0
Total memory bits	65,536 / 3,981,312 (2 %)
Embedded Multiplier 9-bit elements	0 / 532 (0 %)
Total PLLs	0 / 4 (0 %)

Registradores: 1271
DSP: 0
Block memory bits: 65536
ALUTs: 2961

Esses valores indicam que o processador uniciclo implementado utilizou recursos modestos do FPGA, sem demandar DSPs (blocos de multiplicação/soma dedicados) — o que era esperado, já que a ISA reduzida implementada não utiliza multiplicações nem divisões. A

utilização de memória interna está associada às memórias de dados e instruções (ramD e ramI), conforme os arquivos .mif gerados.

Analysis & Synthesis Resource Utilization by Entity

<<Filter>>

	Compilation Hierarchy Node	Combinational ALUTs	Dedicated Logic Registers	Memory Bits	DSP Elements	DSP 9x9
1	|TopDE	2961 (3)	1271 (3)	65536	0	0
1	|Uniciclo:UNI1|	2787 (258)	1156 (32)	65536	0	0
1	|ALU:alu|	206 (206)	0 (0)	0	0	0
2	|ALUControl:aluCtrl|	11 (11)	0 (0)	0	0	0
3	|ControlUnit:ctrl|	9 (9)	0 (0)	0	0	0
4	|ImmGen:gerador|	49 (49)	0 (0)	0	0	0
5	|Registers:banco|	2069 (2069)	992 (992)	0	0	0
6	|ramD:MemD|	92 (0)	66 (0)	32768	0	0
1	|altsyncram:altsyncram_component|	92 (0)	66 (0)	32768	0	0
1	|altsyncram_53l1:auto_generated|	92 (0)	66 (0)	32768	0	0
1	|altsyncram_9db2:altsyncram1|	0 (0)	0 (0)	32768	0	0
2	|sld_mod_ram_rom:mgl_prim2|	92 (72)	66 (57)	0	0	0
1	|sld_rom_sr:\ram_rom_logic_gen:name_gen:info_rom_sr|	20 (20)	9 (9)	0	0	0
7	|ramI:MemC|	93 (0)	66 (0)	32768	0	0
1	|altsyncram:altsyncram_component|	93 (0)	66 (0)	32768	0	0
1	|altsyncram_r5l1:auto_generated|	93 (0)	66 (0)	32768	0	0
1	|altsyncram_keb2:altsyncram1|	0 (0)	0 (0)	32768	0	0
2	|sld_mod_ram_rom:mgl_prim2|	93 (72)	66 (57)	0	0	0
1	|sld_rom_sr:\ram_rom_logic_gen:name_gen:info_rom_sr|	21 (21)	9 (9)	0	0	0
2	|sld_hub:auto_hub|	171 (1)	112 (0)	0	0	0
1	|alt_sld_fab_with_jtag_input:\instrument...h_jtag_input_gen:instrumentation_fabric|	170 (0)	112 (0)	0	0	0
1	|alt_sld_fab:instrumentation_fabric|	170 (0)	112 (0)	0	0	0
1	|alt_sld_fab_alt_sld_fab:alt_sld_fab|	170 (1)	112 (6)	0	0	0
1	|alt_sld_fab_alt_sld_fab_sldfabric:sldfabric|	169 (0)	106 (0)	0	0	0
1	|sld_jtag_hub:\jtag_hub_gen:real_sld_jtag_hub|	169 (132)	106 (78)	0	0	0
1	|sld_rom_sr:hub_info_reg|	19 (19)	9 (9)	0	0	0
2	|sld_shadow_jsm:shadow_jsm|	18 (18)	19 (19)	0	0	0

Aqui temos quantos elementos lógicos, registradores, pinos, blocos de memória etc., foram utilizados. Isso representa os requisitos físicos (área) do nosso design

No Compilation Report, em Timing Analyzer, procuramos por Fmax Summary. Isso nos dá uma estimativa da frequência máxima que nosso design pode operar.

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	101.09 MHz	101.09 MHz	altera_reserved_tck	

Em Timing Analyzer -> Clocks:

Clocks

<<Filter>>

	Clock Name	Type	Period	Frequency	Rise	Fall	Duty Cycle	Divide by	Multiply by	Phase	Offset	Edge List	Edge Shift	Inverted	Master	Source	Targets
1	altera_reserved_tck	Base	100.000	10.0 MHz	0.000	50.000											{ altera_reserved_tck }

Juntamente com Timing Analyzer -> Slow 1200mV 85C Model -> Setup and Hold Summary:

Slow 1200mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	altera_reserved_tck	45.054	0.000

Slow 1200mV 85C Model Hold Summary

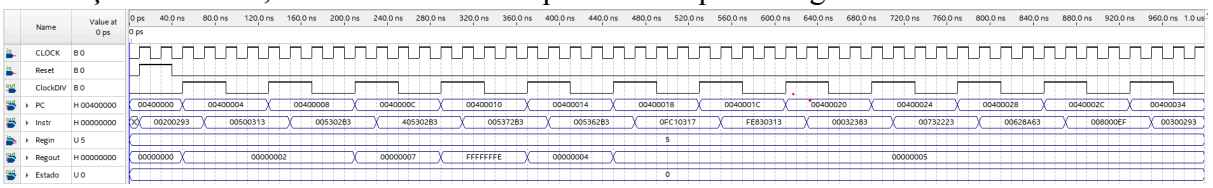
<<Filter>>

	Clock	Slack	End Point TNS
1	altera_reserved_tck	0.402	0.000

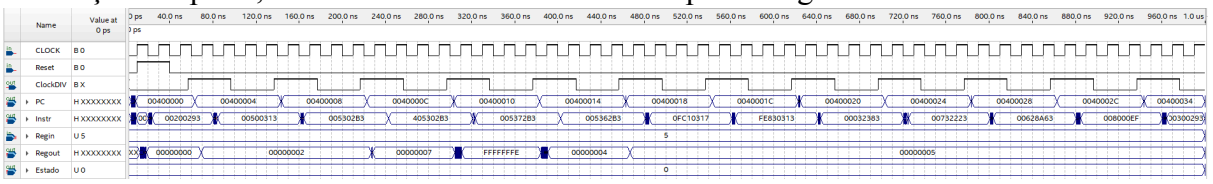
Os Slacks, margens de tempo, são exibidos aqui. Um *slack* negativo significa que o sinal não chega a tempo, indicando uma falha de temporização e que o design não funcionará corretamente na frequência desejada. Um *slack* positivo indica que o requisito de tempo foi atendido, ou seja, o sinal chega antes do tempo de *setup* necessário, e é o que obtemos no nosso projeto.

c) As simulações funcional e temporal foram realizadas utilizando formas de onda para observar o comportamento do processador durante a execução do programa de1.s. Com os arquivos de1_text.mif e de1_data.mif gerados pelo RARS configuramos o arquivo de teste.

Simulação funcional, onde é considerado que todas as portas lógicas não tem atraso.



Simulação temporal, onde é considerado o atraso nas portas lógicas.



Verificamos que as formas de onda correspondem ao comportamento esperado do programa de1.s. O PC inicia em 0x00400000 e incrementa de 4 em 4, indo para 00400004, 00400008, etc. Isso mostra que o processador está buscando instruções sequencialmente, como esperado em uma CPU uniciclo.

A cada avanço do PC, Instr assume um novo valor hexadecimal, como:

- 00200293 → addi t0, zero, 2
- 00500313 → addi t1, zero, 5
- 005302B3 → add t0, t1, t0
- 405302B3 → sub t0, t1, t0

Essas instruções batem exatamente com as primeiras do de1.s, o que confirma que a memória de instruções (ramI) está funcionando.

Regin = 5 → indica que estamos observando o registrador t0 (x5).
Regout mostra os valores de t0 durante o programa:

- 00000007 → após t0 = 2 + 5 = 7 (add)

- FFFFFFFE → resultado da subtração: 5 - 7 = -2 (em complemento de dois: 0xFFFFFFFE)
- 00000004 → or com 5 | -2 provavelmente resultando em 0xFFFFFFFF, e depois sobrescrito.

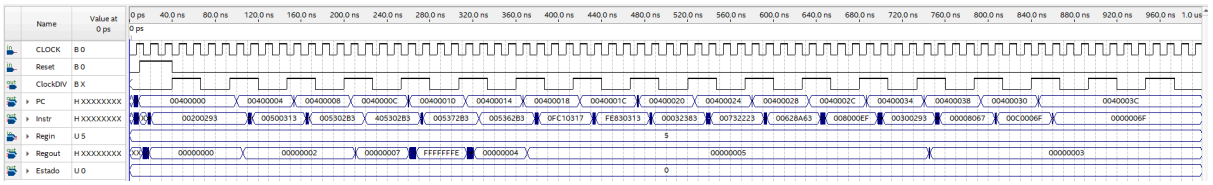
O registrador t0 está sendo corretamente carregado, somado, subtraído, etc e t1 está sendo carregado com 5, e usado nos cálculos.A instrução lw está em FE830313 e sw em 00732223 — o que mostra que elas foram decodificadas e executadas. E conseguimos ver a leitura do valor 5 e escrita em valor+4.

A instrução jal aparece como 008000EF — isso corresponde a jal ra, FUNCAO, depois a jalr também é vista (00300293, jalr zero, 0(ra)). Isso mostra que o controle de fluxo está sendo executado corretamente.

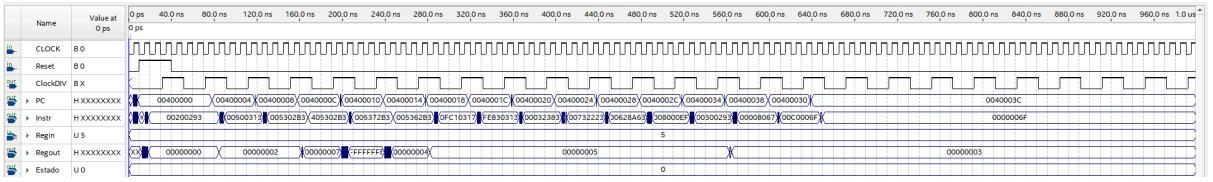
d)

Realizando simulações temporais de forma de onda com diferentes frequências de clock, obtivemos os seguintes resultados.:

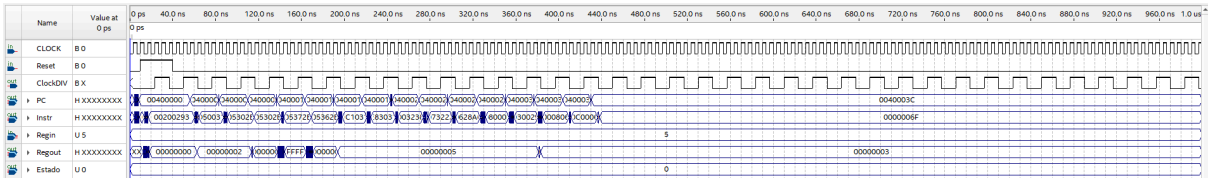
75MHz - Processador funcionando normalmente



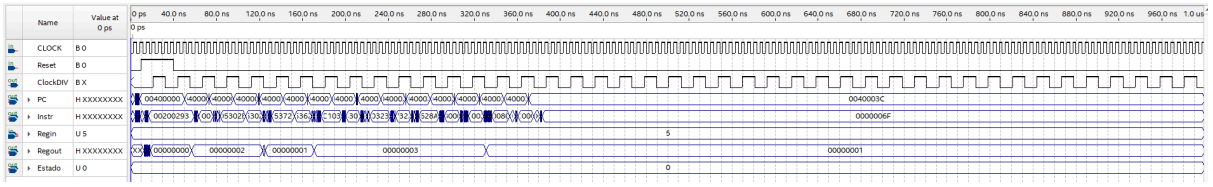
100MHz - Processador funcionando normalmente



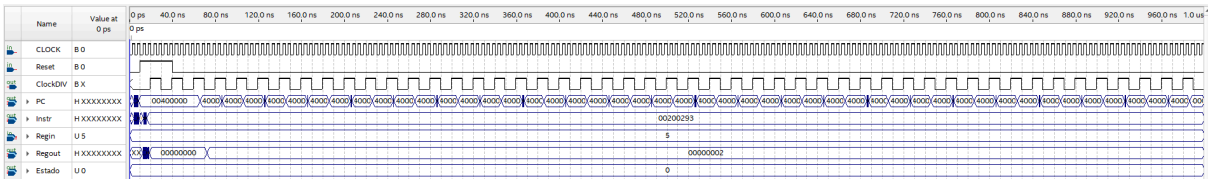
150MHz - Processador funcionando normalmente



175MHz - Processador retornando saídas erradas



200 MHz - Processador não computa as instruções corretamente



Com esses resultados, podemos perceber que por mais que a simulação do quartus indique a frequência máxima como 101.09MHz, o processador consegue operar sem grandes problemas até aproximadamente 150MHz.