Universidade de Brasília
Instituto de Exatas
Departamento de Ciência da Computação

Relatório do Laboratório 2
Processador Uniciclo

CIC0099 - Organização e Arquitetura de Computadores

Turma 02 - Prof. Dr. Ricardo Pezzuol Jacobi

Grupo B4

Henrique Morcelles Salum	232003008
Athos Calixto Muniz	211068261
Gabriela Fernanda Rodrigues Costa	180120859
Cauê Araújo Euzebio	211028195
Lucas Silva Nóbrega	180035096

Sumário

1 Introdução

O presente experimento tem como objetivo introduzir os alunos ao processo de projeto e implementação de sistemas digitais utilizando VHDL como Linguagem de Descrição de Hardware (HDL). A atividade busca promover o treinamento prático em modelagem, análise e síntese de circuitos digitais, utilizando o ambiente de desenvolvimento **Intel Quartus Prime v24.1** como ferramenta principal.

Como aplicação prática, propõe-se a implementação de um processador Uniciclo compatível com a ISA RV32I reduzida, contendo as instruções:`add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `beq`, `jal`, `jalr`, `addi` e `lui`. Serão desenvolvidos os blocos fundamentais do processador, como banco de registradores com três portas de leitura, gerador de imediatos, ULA e unidade de controle.

A implementação será validada por meio de simulação funcional e temporal, com análise de *timing* e frequência máxima de operação, permitindo compreender de forma integrada o funcionamento de uma arquitetura RISC-V.

2 Métodos e Análise

2.1 Experimento

Enunciado

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

Para realizar o solicitado, é-nos fornecido o esquemático de uma implementação do Uniciclo com ISA reduzida. Nesse esquemático, porém, foi necessário realizar alterações no caminho de dados, que serão devidamente justificadas adiante.

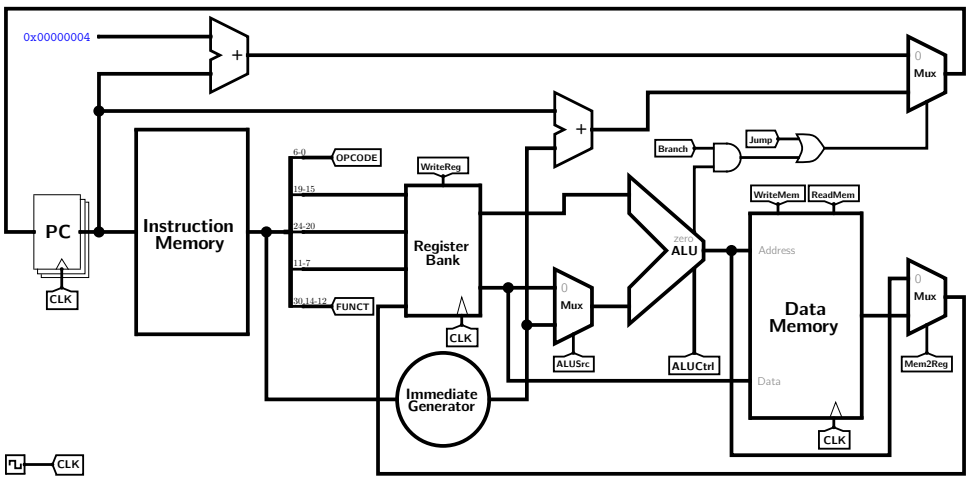

Figura 1: Esquemático do Uniciclo com ISA reduzida

Em que os túneis representam entradas e saídas do módulo de controle do processador, que foi abstraído na figura acima.

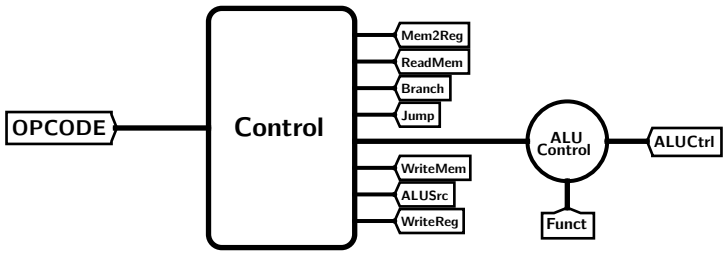

Figura 2: Módulo de controle do Uniciclo com ISA reduzida

2.1.1 Tarefa I

Enunciado

Analise o programa `de1.s` que testa a corretude da implementação de todas as $9 + 3$ instruções e teste no RARS. *Dica: O registrador `t0` é usado para visualizar resultados!*

O programa `de1.s`, que testa a corretude da implementação das instruções do RISC-V no RARS, ou seja, verifica se as instruções são executadas corretamente, foi testado utilizando a execução passo a passo, e os resultados foram visualizados por meio do registrador `t0`, de acordo com o que foi sugerido. Os registradores `t1` e `t2` são os registradores temporários utilizados para as operações e `gp` é o ponteiro para a área de dados, inicializado em `0x100100000`.

O programa testa as instruções `add`, `sub`, `and`, `or`, `slt`, `lw`, `sw`, `beq`, `jal`, `jalr`, `addi` e `lui`.

Instrução	Código	Entrada	Saída
lui	`lui gp,0x10010`	`gp, imm = 0x10010`	`gp = 0x10010000`
lw	`lw t1,0(gp)`	`gp = 0x10010000, mem[0x10010000] = 0xFFFFFF0F`	`t1 = 0xFFFFFF0F`
addi	`addi t2,zero,0x777`	`zero = 0x00000000, imm = 0x777`	`t2 = 0x00000777`
and	`and t0,t1,t2`	`t1 = 0xFFFFFF0F, t2 = 0x00000777`	`t0 = 0x00000707`
or	`or t0,t1,t2`	`t1 = 0xFFFFFF0F, t2 = 0x00000777`	`t0 = 0xFFFFFF7F`
add	`add t0,t2,t1`	`t2 = 0x00000777, t1 = 0xFFFFFF0F`	`t0 = 0x00000686`
sub	`sub t0,t2,t1`	`t2 = 0x00000777, t1 = 0xFFFFFF0F`	`t0 = 0x00000868`
slt	`slt t0,t1,t2`	`t1 = 0xFFFFFF0F, t2 = 0x00000777`	`t0 = 0x00000001`
slt	`slt t0,t2,t1`	`t2 = 0x00000777, t1 = 0xFFFFFF0F`	`t0 = 0x00000000`
beq	`beq t0,zero,PULA`	`t0 = 0x00000000, zero = 0x00000000`	Salta para `PULA`
jal	`jal PROC`	`PC atual, ra = PC + 4`	Chama `PROC`, `ra = end. retorno`
sw	`sw t0,4(gp)`	`t0 = 0x0000007F, gp = 0x10010000`	`mem[0x10010004] = 0x0000007F`

Tabela 1: Código e saídas associadas

Uma explicação resumida do código: Inicializa-se o ponteiro de memória carregando o endereço `0x10010000` no registrador `gp`, e carrega-se o valor `0xFFFFFF0F` da memória

para `t1` e inicializa-se `t2` com `0x777`. Então, é executado várias operações lógicas e aritméticas com `t1` e `t2`, AND e OR, soma (`add` e `addi`) e subtração (`sub`), comparações (`slt`) que verificam se um valor é menor que o outro, teste desvio condicional e chamada de subrotina.

2.1.2 Tarefa II

Enunciado

Implemente o Banco de Registradores com 3 leituras simultâneas: `rs1`, `rs2` e `disp`.
Stack Pointer (`sp`) inicial: `0x1001_03FC` Global Pointer (`gp`) inicial: `0x1001_0000`

O Banco de Registradores é um componente fundamental e funciona como a memória mais rápida e acessível do processador. O programa com 3 leituras simultâneas, `rs1`, `rs2` e `disp`, já estava quase todo implementado em VHDL (xreg.vhd) - faltava apenas atribuir ao `oREGD` (o `disp` do enunciado) o valor no registrador da posição `iDisp` do banco de registradores.

O banco de registradores é iniciado (e reinciado, quando a entrada `iRST` está habilitada) com todos os registradores zerados, com exceção do `sp` e do `gp`, que têm os valores definidos no enunciado. Esses valores representam, respectivamente, o endereço da pilha de memória e a área de variáveis globais. Essas definições foram feitas no arquivo riscv_pkg.vhd.

A operação ocorre em dois modos: atualização **assíncrona** dos registradores de saída `oREGA`, `oREGB` e `oRegD` nos endereços `iRS1`, `iRS2` e `iDISP`, respectivamente; e escrita **síncrona** no banco de registradores (a não ser que o *reset* esteja habilitado, nesse caso, são escritos os valores do estado inicial nos registradores de forma **assíncrona**). Quando a CPU precisa guardar o resultado de algum cálculo ou armazenar algum valor da memória, o sinal `iWREN` é ativado e, no endereço do registrador destino `iRD`, é escrito o dado em `iDATA`.

2.1.3 Tarefa III

Enunciado

Implemente o Gerador de Imediatos.

O Gerador de Imediatos, também já quase todo implementado (gen_imm.vhd), recebe 32 bits da instrução e gera um imediato de 32 bits. Este componente é responsável por extrair e processar os valores imediatos presentes nas instruções, considerando que os campos de imediatos não possuem um padrão uniforme. A implementação consiste em um case, que é sintetizado em um multiplexador, em que dependendo do opcode (o tipo da instrução (I, S, B, U, J)) geramos imediatos com procedimentos distintos.

Inicialmente, é necessário analisar os 7 bits menos significativos da instrução, que são aqueles que correspondem ao campo opcode e vão servir como um seletor para determinar qual o formato de instrução que está sendo processado.

O primeiro caso, é das instruções Tipo-I, que, no nosso projeto, agrupam os opcodes `OPC_LOAD`, `OPC_OPIMM`, `OPC_JALR`. Nesse caso, o imediato tem 12 bits (os bits 31 a 20 da instrução), precisamos apenas extendê-lo para 32 bits.

O segundo caso tratato são as instruções Tipo-S (*store*). Nelas o imediato é dividido em duas partes não contíguas na instrução, os bits 31 a 25 são os mais significativos, e os bits 11 a 7 são os menos significativos, gerando-se assim os 12 bits. O resultado da concatenação é estendido para 32 bits.

O terceito caso consiste nas instruções Tipo-B (`branch`), que possuem uma estrutura mais complexa que as anteriores. Seus campos estão espalhados em quatro posições diferentes na instrução, do mais ao menos significativo: o bit 31, bit 7, bits 30 a 25 e 11 a 8. Eles são concatenados nessa sequência e, então, é adicionado um '0' no bit menos significativo, já que, pela forma como é organizada a memória, as instruções ficam em endereços pares. O resultado dessa sequência de manipulações é um sinal com 13 bits, que precisa ser estendido para 32.

O quarto caso, as instruções Tipo-J (`jal`), é similar ao anterior: os bits estão espalhados e são reordenados, além de ser concatenado à direita o bit '0'. A diferença é que as instruções de Tipo-J dedicam 20 bits ao imediato, de forma que, com a concatenação com 0, já são 21 bits, e o resultado só precisa ser estendido em 11 bits.

O quinto e último caso de interesse (pois, nos restantes, o imediato é um conjunto de *don't cares* que arbitramos como 32 zeros) é o das instruções Tipo-U. Nesse caso, na instrução há 20 bits contíguos reservados ao imediado; basta estender essa sequência para 32 bits e pronto. Esse caso não estava tratada no esqueleto de projeto fornecido.

2.1.4 Tarefa V

Enunciado

Implemente a ULA mínima necessária (`add`, `sub`, `and`, `or`, `slt`, `zero`).

A ULA (Unidade Lógica e Aritmética) é um componente fundamental responsável por realizar operações tanto lógicas quanto aritméticas; sendo elas, no caso deste laboratório: adição, subtração, *bitwise AND*, *bitwise OR* e SLT (Set if Less Than). Adicionalmente, ela deve ser capaz de fazer o 'OU' entre todos os bits de uma palavra, pois isso é necessário para calcular a saída 'zero'.

Na implementação desse módulo, seguimos um estilo híbrido entre estrutural e comportamental: ao mesmo tempo que instanciamos um somador para realizar somas e subtrações, utilizamos abstrações de alto nível, nesse caso, as estruturas `case` e `when ... else`, para implementar o `slt`, `or`, `and` e calcular a saída `zero`.

A ULA implementada recebe como entrada os valores dos dois operandos e um sinal de 4 bits `ALUCtrl`, que indica qual das operações previamente citadas deve ser realizada. Dessa forma, a implementação é dada a partir de um 'case' no qual, para cada código recebido pelo sinal "ALUCtrl", é realizada uma operação lógica ou aritmética.

Por último, a ULA tem dois sinais de saída: um para o resultado da operação realizada e outro para o sinal `zero`, que é ligado caso o resultado da operação seja '0' - ele é usado especificamente no caso do `branch`, se a subtração dos operandos for '0', o 'zero' é ligado o posterior AND desse com sinal com o sinal de controle 'branch' faz com que o endereço do salto seja escrito no PC.

2.1.5 Tarefa VI

Enunciado

Implemente o Controlador da ULA e o Bloco Controlador.

O Bloco Controlador é o verdadeiro "maestro" do processador, ou seja, ele é responsável por gerar os sinais de controle necessários para gerenciar todo o caminho de dados e para coordenar quase todos os outros componentes da CPU, entre eles: a memória de dados, o banco de registradores, o controlador da ULA e vários multiplexadores.

Esse módulo recebe como entrada o 'opcode' (código de operação), um sinal de 7 bits que determina qual é a operação básica que implementa a instrução. Considerando a ISA reduzida, são possíveis oito 'opcodes': um para todas as instruções Tipo-R e mais uma para cada uma das outras instruções (`sw`, `lw`, `addi`, `beq`, `jalr`, `jal` e `lui`).

Além disso, o Bloco de Controle, envia, como saída, os sinais que coordenam os outros componentes do processador: para cada 'opcode', o Bloco de Controle determina esses sinais para controlar exatamente como o resto do processador deve agir para que a instrução recebida seja, de fato, executada. Seguem os sinais de saída:

- `mem2Reg` - 1 bit - Indica qual sinal deve estar disponível para ser escrito no banco de registradores. Caso '0': resultado da ULA. Caso '1': saída da memória de dados;
- `memRead` - 1 bit - Determina se há necessidade de leitura na memória de dados. Caso '0': desabilita a leitura. Caso '1': habilita a leitura;
- `branch` - 1 bit - Determina se PC deve receber PC + 4 ou PC + imediato. Caso '0': PC + 4. Caso '1': PC + Imm;
- `jump` - 1 bit - Determina se haverá salto incondicional. Caso '0': desabilita o salto. Caso '1': habilita o salto;
- `ALUOp` - 2 bits - Determina a entrada do Controlador da ULA;

- `memWrite` - 1 bit - Determina se a escrita na memória de dados deve estar habilitada. Caso '0': desabilita. Caso '1': habilita;
- `ALUSrc` - 1 bit - Indica a origem do segundo operando da ULA. Caso '0': sinal com origem no banco de registradores. Caso '1': sinal com origem no gerador de imediatos;
- `regWrite` - 1 bit - Determina se a escrita no banco de registradores deve estar habilitada. Caso '0': desabilita. Caso '1': habilita.

A implementação do Bloco de Controle relaciona sua entrada com suas saídas de acordo com a tabela a seguir:

Saída / Opcode	mem2Reg	memRead	branch	jump	ALUOp	memWrite	ALUSrc	regWrite
OPC_RTYPE	0	0	0	0	10	0	0	1
OPC_OPIMM	0	0	0	0	10	0	1	1
OPC_STORE	X	0	0	0	00	1	1	0
OPC_LOAD	1	1	0	0	00	0	1	1
OPC_BRANCH	X	0	1	0	01	0	0	0
OPC_JAL	X	0	0	1	XX	0	X	1
OPC_JALR	0	0	0	1	00	0	1	1
OPC_LUI	0	0	0	0	00	0	1	1

Tabela 2: Sinais de controle gerados para cada opcode

Agora, o Controlador da ULA aparece dado que o projeto tem formato hierárquico, ou seja, o Bloco de Controle coordena o Controlador da ULA, que, por sua vez, coordena a própria ULA. Assim, o Bloco de Controle pode receber apenas o 'opcode' enquanto o Controlador da ULA fica com o serviço de receber os campos 'funct7' e funct3', que especificam o tipo de operação aritmética/lógica. Dessa forma, o Controlador da ULA recebe o sinal 'ALUOp', previamente visto, e, a partir do mesmo e dos sinais 'funct7' e 'funct3', ele consegue determinar para a ULA exatamente qual operação deve ser executada através do sinal de saída 'ALUCtrl'. Por conseguinte, o sinal 'ALUOp' identifica cada uma das seguintes categorias de instruções:

Acesso à memória (lw, sw) - "00" - [add] Desvio (branch) - "01" - [sub] Lógico-aritméticas - "10" - [add, sub, and, or, slt] Então, caso o sinal 'ALUOp' seja "00", o Controlador da ULA determina para a ULA que a operação realizada deverá ser uma soma, da mesma forma que, caso o sinal 'ALUOp' seja "01", será determinada uma subtração e, por último, caso seja "10", a partir dos sinais 'funct7' e 'funct3', será determinada a operação a ser realizada.

3 Conclusão