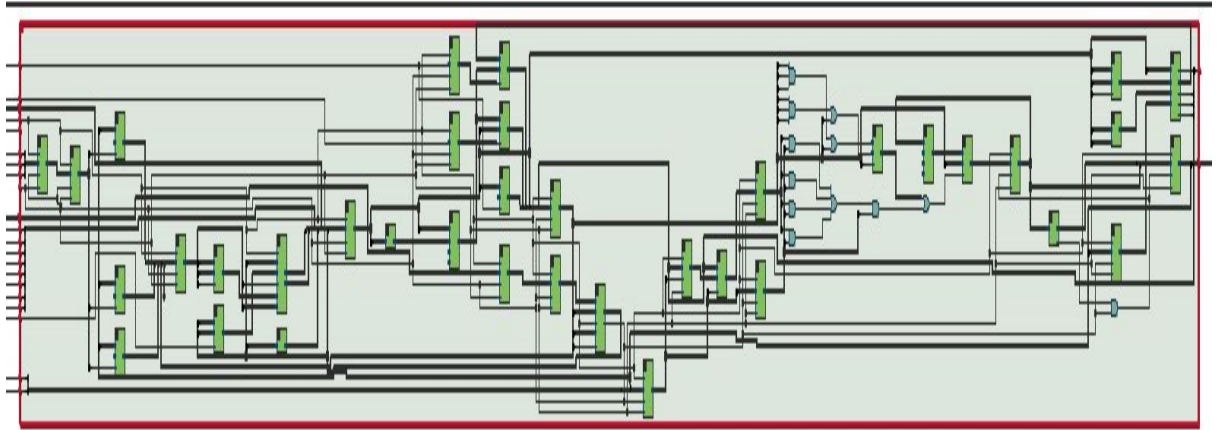


Implementação do Conjunto de Instruções LEGv8

Grupo E



1. Descrição da tarefa	3
2. Decisões de Projeto	6
2.1 Suporte a diversos tipo de Branch	6
B.Cond	7
BL	8
Branch to Register	8
2.2 Suporte a formatos de 8, 16 e 32 bits	9
Load	9
Store	10
2.3 Instruções Aritméticas	10
2.4. Interface com unidade de Ponto Flutuante	14
2.5 Interface com o módulo de memória	15
2.6. Projeto da Unidade de Controle Completa	16
3. Dependências com outros grupos	17
4. Mockups	18
Resultados e Simulações	19
Conclusões	21

1. Descrição da tarefa

A tarefa do grupo consistiu na implementação do subconjunto de instruções do ARMv8, o LEGv8. A descrição do conjunto de instruções completa é apresentada em [1]. Utilizamos esta referência por conter a descrição completa do comportamento esperado para cada uma das instruções. Já o subconjunto LEGv8 é apresentado por Patterson em [2], descrevendo um subconjunto Turing-completo das instruções ARM, com algumas simplificações didáticas.

A construção do projeto será feita a partir do fluxo de dados e unidade de controle apresentados em [2]. A versão final do pipeline apresentado no livro está na Figura 1.

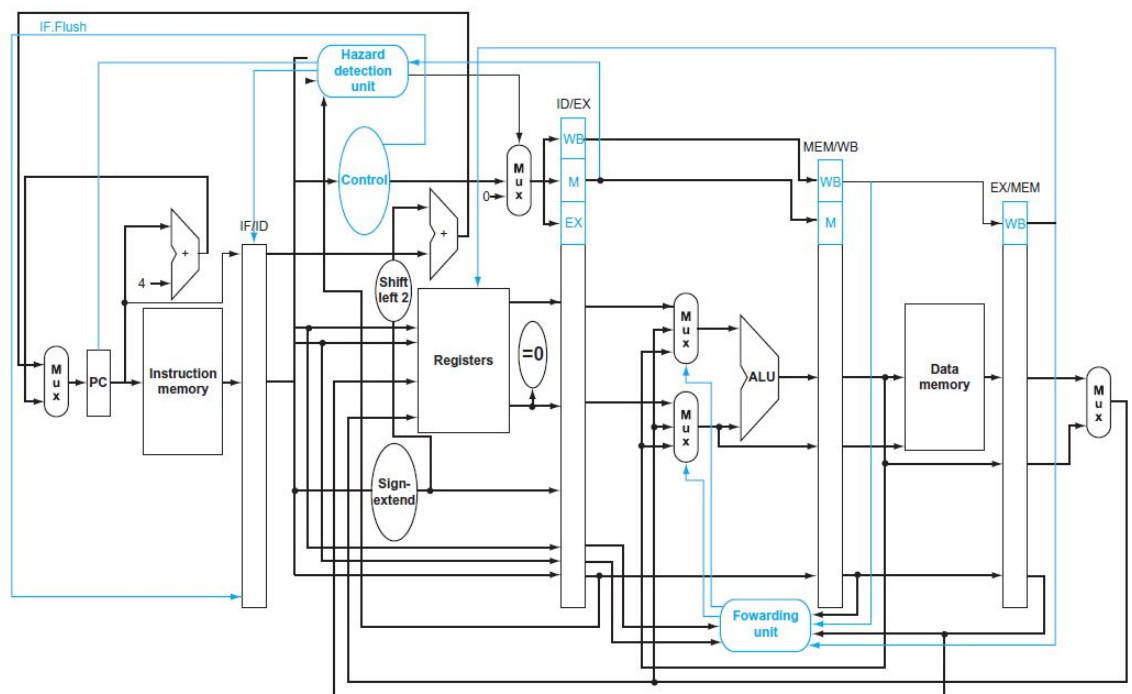


Figura 1: Datapath final, conforme extraído de [2].

Esse fluxo de dados, contudo, implementa um sistema de detecção de hazard e forwarding. Como outros grupos tiveram a tarefa de implementar essas unidades, elas não foram feitas nesse projeto. Além disso, esse DP transfere a tomada da decisão de branch para o estágio de instruction decode, que também não foi feita pois fugia do escopo.

A partir dessa implementação simplificada, o fluxo de dados e a unidade de controle foram expandidos para suportar todo o conjunto de instruções requerido. Isso foi feito a partir da descrição do comportamento necessário para a execução da instrução. Um sumário das alterações é:

- Implementado formas de lidar com dados de 8, 16, 32 bits tanto no acesso a memória tanto para leitura (LOAD) quanto para escrita (Store);

- Implementado hardware para realizar as instruções de MOV, incluindo forma de preservar valor antigo no registrador ou preencher com zeros. Isso foi feito na ULA, colocando o valor antigo no operando A e o valor estendido de 16 bits, além dos dois bits de codificação do shift no operando B.
- Implementado hardware para instruções de branch and link, que precisa salvar o conteúdo de PC num registrador e BR, para carregar o conteúdo de um registrador no PC. Foi feito com um multiplexador.
- Implementada lógica para lidar com flags na ALU. Foi criada a saída das Flags na unidade e que são salvas quando o sinal Flags da unidade de controle está ativo. Além disso, foi criada a lógica para implementar os branches com as flags, por meio de um conjunto de portas lógicas and e or.
- Adicionadas funções adicionais a ALU, como por exemplo ORR, EOR, SLL, SR, totalizando dez operações junto com os dois moves..
- Implementada comunicação com a unidade de ponto flutuante. Feito passando-se os operandos e o campo shamt, para que a unidade
- Implementado tratamento de “miss” dos caches de dados e instrução. Rudimentar, apenas desabilitando o enable dos registradores do pipeline.
- Implementados novos campos de extensão de operador imediato, para suportar todos os formatos de instrução e variantes, como o B.Cond, que é uma variação de B.

Como a arquitetura ARM é proprietária, não há bons exemplos de implementação de processadores completos. Contudo, dado a similaridade com outras arquiteturas RISC, como o MIPS e o RISC-V, é possível reaproveitar ideias desses projetos. Observamos e adotamos a metodologia de projeto de hardware conforme o apresentado em [3], que consiste em dividir o projeto do sistema em três principais descrições: funcional, estrutural e física. Além disso, também ocorre divisão do projeto em quatro níveis de abstração: Processador, Transferência de Registradores, Componentes Lógicos e Transistores.

Nos foi apresentada uma descrição funcional do processador, a sua arquitetura do conjunto de instruções, com o requisito não funcional de ser implementado em uma organização pipeline. Com isso, é possível obter uma descrição estrutural com os componentes do processador, exibida na Figura 1 de forma simplificada. Em seguida, fez-se uma descrição funcional de cada uma das unidades funcionais do processador. Essa descrição, sendo feita em VHDL, foi convertida em uma descrição em nível de transferência de registradores. Após obter e validar essa descrição, definindo-se como alvo a placa Cyclone-V, é possível utilizar o QUARTUS para mapear o RTL em componentes lógicos e por fim em unidades funcionais da FPGA. As unidades funcionais da FPGA, por sua vez, já tem sua implementação feita a nível de transistores. Assim, escolhe-se para esse projeto a utilização de uma metodologia Top-Down de projeto de hardware, baseado no método 2.5 apresentado em [3]. Esse método é sumarizado na Figura 2.

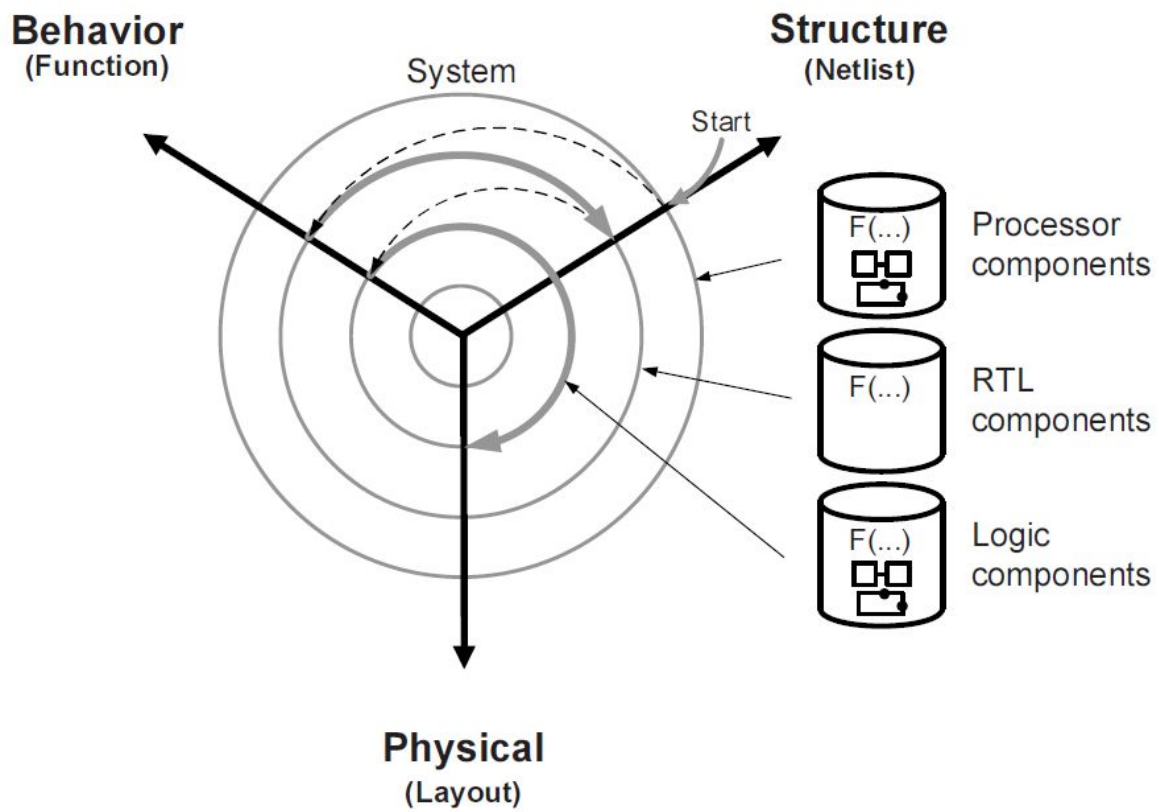
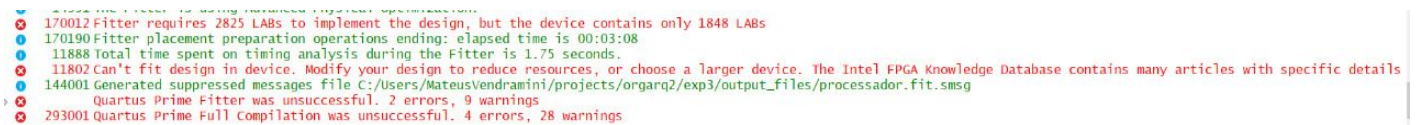


Figura 2: Sumário da metodologia de projeto baseada em FPGA. Extraído de [3].

2. Decisões de Projeto

Dado o que foi apresentado no primeiro capítulo, devido a ausência de documentação de implementações abertas desse conjunto de instruções, devido a sua vantagem comercial, adotou-se neste projeto uma abordagem “as simple as possible”, criando as modificações na estrutura do fluxo de dados de forma a gerar o menor impacto possível no hardware que já estava descrito.

Esta decisão resultou em um hardware que implementa todas as funcionalidades requisitadas, mas de forma ineficiente. A maior evidência disso é que o projeto final, apesar de ser sintetizável, não o pode ser feito na placa Cyclone-V por falta de capacidade da mesma. Essa falta de capacidade é evidenciada na Figura 3.



```
170012 Fitter requires 2825 LABs to implement the design, but the device contains only 1848 LABs
170190 Fitter placement preparation operations ending: elapsed time is 00:03:08
11888 Total time spent on timing analysis during the Fitter is 1.75 seconds.
11802 Can't fit design in device. Modify your design to reduce resources, or choose a larger device. The Intel FPGA Knowledge Database contains many articles with specific details
144001 Generated suppressed messages file C:/Users/MateusVendramini/projects/orgarq2/exp3/output_files/processor.fit.smsg
Quartus Prime Fitter was unsuccessful. 2 errors, 9 warnings
293001 Quartus Prime Full Compilation was unsuccessful. 4 errors, 28 warnings
```

Figura 3: Falha na geração de rotas para o projeto por falta de capacidade.

Ao longo desse documento serão descritas as modificações realizadas ao pipeline original. As modificações podem ser divididas em duas grandes tarefas:

- Expansão da unidade de controle, para suportar todas as instruções.
- Adição de multiplexadores e novos campos nos registradores de estágio para suportar as novas operações.

Conforme apresentado no cronograma inicial, as implementações a serem realizadas eram:

1. Implementação de todas as instruções de Branch
2. Implementação de instruções com vários tipos de dados (8, 16, 32)
3. Implementação de todas as instruções aritméticas
4. Implementar chamadas da unidade de ponto flutuante

Assim, os próximos capítulos descreveram, para cada um desses conjuntos de instrução, os novos sinais de controles adicionados e uma descrição breve das alterações no pipeline realizadas para suportar essas operações. São destacados os estágios em que ocorreram a maior parte das alterações. Contudo, toda adição envolve a extensão dos registradores de estágio, adicionando novos campos.

2.1 Suporte a diversos tipo de Branch

Consultando o greencard, as instruções de branch previstas são:

- Branch incondicional (B)
- Branch condicional (B.cond)
- Branch with Link (BL)
- Branch to Register (BR)
- Compare & Branch if not zero (CBNZ)
- Compare & Branch if zero (CBZ)

Das instruções dessa lista, CBZ, CBNZ e B estavam no primeiro datapath descrito pelo livro, com os sinais de controle:

- Uncondbranch: Branch incondicional. Se igual a um, branch é executado
- Branch: Se igual a um, sinal de zero da ula, obtida no estágio de ex, ligado e CBZ em zero, desvia.
- CBZ: Semelhante a branch, se Branch='1', CBZ='1' e ZERO = '0', desvia

Além disso, a unidade de extensão de zeros estende:

- Instr[25,0] copiando o valor de Instr[25] para os 38 bits restantes para a instrução de branch incondicional
- Instr[23,5], copiando o valor de Instr[23] para os 45 bits restantes para a instrução de CBZ e CBNZ.

E o fluxo de dados que suporta essas instruções é exatamente igual ao apresentado no final do livro.

A seguir, serão descritas as modificações e sinais de controle dos novos branches. Nessas seções “Instr” se refere a instrução de 32 bits da arquitetura. M[reg] se refere ao conteúdo do registrador. É suposto que o leitor tem conhecimento dos 5 estágios de pipeline para o conjunto LegV8, descrito em [2].

B.Cond

Para a instrução de B.cond, segundo o manual de referência da ARM [1], os últimos 4 bits codificam as flags de condição a serem checadas:

- Instr[3]: Overflow: Desvia se flag de overflow foi setada.
- Instr[2]: Negative: Desvia se flag de negative foi setada.
- Instr[1]: Carry: Desvia se flag de carry in foi setada.
- Instr[0]: Zero - Desvia se flag de zero foi setada.

A decisão de desvio é tomada se o bit bcond = '1' e existe algum sinal de B.cond que tem valor um setado nas flags salvas no processador.

As flags são salvas pelas instruções SETFlags descritas na seção sobre instruções aritméticas.

A decisão do desvio é tomada no estágio de acesso a memória e foi implementada diretamente com portas lógicas:

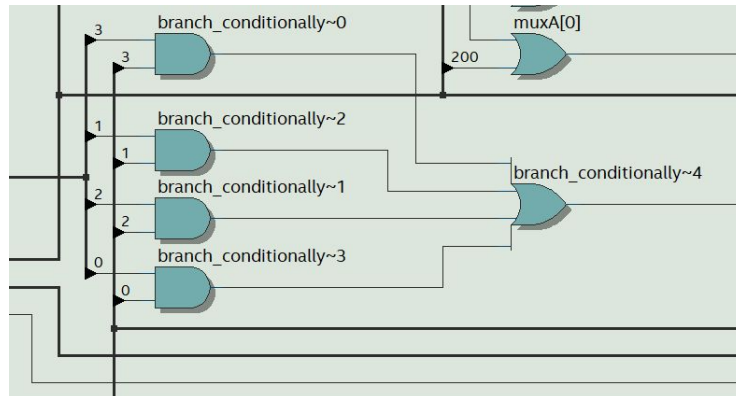


Figura 3: Tomada de decisão no estágio de memória. Por meio da porta or com todos os sinais, qualquer um dos sinais em high leva ao desvio.

Também foi expandida a unidade de extensão de sinal do estágio ID:

- Instr[23,4] copiando o valor de Instr[23] para os 44 bits restantes.

BL

Para a instrução de Branch with Link são necessárias as seguintes ações:

- $M[30] \leftarrow PC_{atual} + 4$
Ambas essas modificações podem ser feitas com três mux cada, da seguinte forma:
- Mux para selecionar entre o número do registrador destino e 31, com blink = '1' selecionando 30.
- Mux para selecionar entre o resultado da ALU e $PC + 4$, com blink = '1' selecionando $PC + 4$.

Assim, todas as modificações podem ser feitas no estágio de execute. Não foi necessário modificar a unidade de extensão pois o formato da instrução é igual ao de Branch incondicional.

Branch to Register

A instrução de Branch to Register consiste em uma ação:

- $PC \leftarrow M[Rt]$

Desse modo, ela funciona como um branch incondicional, com a diferença de que o endereço não é calculado a partir do PC atual e sim do valor lido no registrador Rt. Seguindo a linha da instrução anterior, basta um mux para resolver o problema:

- Mux para selecionar entre NPC calculado e M[Rt], com bregister = '1' selecionando M[Rt].

2.2 Suporte a formatos de 8, 16 e 32 bits

O suporte a esses dados feito nas instruções de Load e Store. Essa seção então está dividida para cada uma dessas adaptações:

Load

As instruções de Load previstas pelo greencard são:

- Load Register Unscaled offset (64 bits)
- Load Byte Unscaled offset (8 bits)
- Load Half Unscaled offset (16 bits)
- Load Word Unscaled offset (32 bits)

Como os registradores da arquitetura são de 64 bits, o manual diz que os bits não carregados da memória devem ser completados com zero (zero extension). Para isso, foi criado o sinal de controle:

- zeroext0
- zeroext1
- zeroext2

Para as codificações, temos:

- zeroext0='0'; zeroext1='0' e zeroext2='0': 64 bits
- zeroext0='0'; zeroext1='0' e zeroext2='1': 32 bits
- zeroext0='0'; zeroext1='1' e zeroext2='1': 16 bits
- zeroext0='1'; zeroext1='1' e zeroext2='1': 8 bits

A extensão de zeros é feita por meio de 3 multiplexadores, exibidos na Figura 4 a seguir.

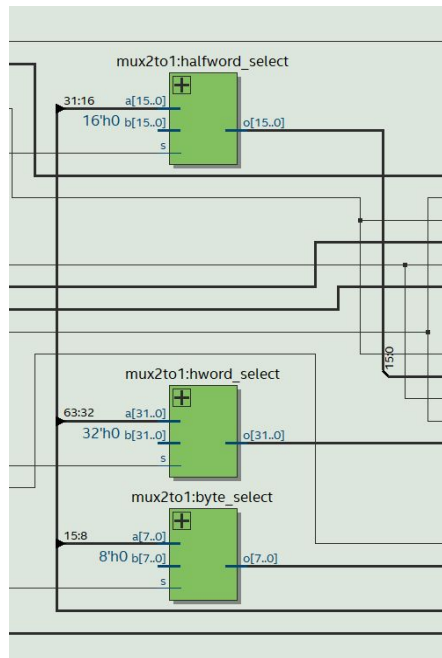


Figura 4- Extensão de zeros para instruções de store.

Caso o sinal de zero correspondente esteja em High, é colocada com saída um vetor de zeros. Desta maneira, todas as alterações ficam concentradas no estágio de memória, após o valor ser recebido da memória.

Store

Para a instrução de store, foi necessário modificar a interface com o módulo de acesso a memória, já que a memória precisa ser capaz de escrever valores de 8, 16, 32 e 64, endereçados a nível de byte. Assim, foi criado um novo sinal, numBytes, que codifica essas escritas:

- numBytes = “00”: escreve valor de 64 bits
- numBytes = “01”: escreve valor de 32 bits
- numBytes = “10”: escreve valor de 16 bits
- numBytes = “11”: escreve valor de 8 bits
-

A interface final com a memória é exibida na Figura 5, apresentada adiante.

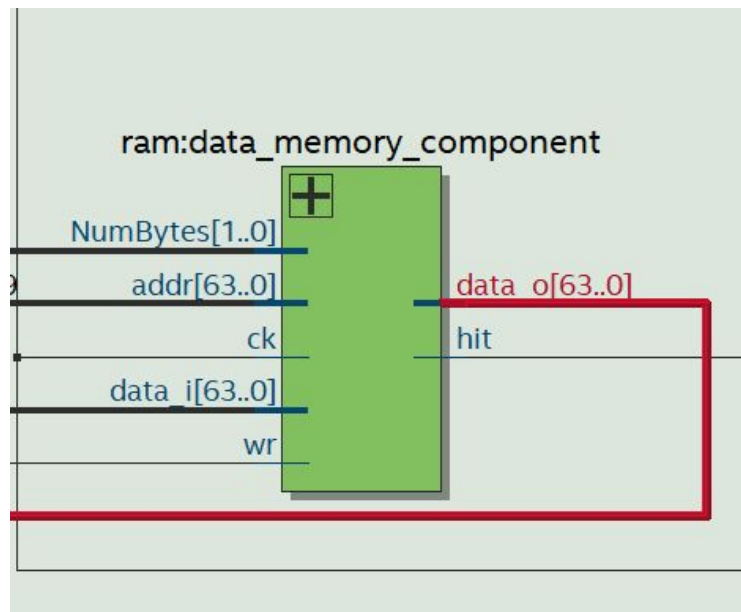


Figura 5: Interface com a memória.

2.3 Instruções Aritméticas

Para a extensão das necessidades aritméticas foi necessário:

- Expandir as flags setadas pela ULA, além do zero, carry in, overflow e negativa.
- Expandir a unidade de controle da ULA para decodificar todas as instruções
- Suportar as novas operações, como MOVk, MOVz, ORR, etc.

Tanto as flags quanto as novas operações foram feitas diretamente dentro do componente da ALU. A interface final do componente é apresentada abaixo:

```
entity alu is
  port (
    A, B      : in  signed(63 downto 0);    -- inputs
    F         : out bit_vector(63 downto 0); -- output
    S         : in  bit_vector(3 downto 0);  -- op selection
    Over      : out bit;    --overflow flag
    Negative  : out bit;    --negative flag
    Carry     : out bit;    --carry flag
    Z         : out bit;    -- zero flag
    shift_amount_ex : in bit_vector(5 downto 0)
  );
end entity alu;
```

As flags são registradas num registrador especial localizado no estágio de execução e que tem como enable o novo sinal da unidade de controle, o SetFlags. Assim, sempre que ele está habilitado, as flags da ULA são salvas nesse registrador, ao final do ciclo. Para manter o pipeline consistente, a saída deste registrador é adicionada ao registrador de estágio EX/MEM, onde a decisão do jump é tomada.

Para as operações, a versão final da ULA suporta, com seus códigos de operação apresentados na Tabela 1:

Operação	Selection
AND	0000
OR	0011
ADD	0010
SUB	0110
Copy A	0111
NOR	1100
XOR	0001
Shift Right	1001
Shift Left	1000
Mov z	0100
Mov k	0101

Tabela 1: Operações suportadas pela ULA e respectivos códigos de seleção

As operações básicas foram descritas utilizando diretamente construções da biblioteca `numeric_bit`. Cabe detalhar um pouco mais as operações de Mov.

Para essas instruções, o formato é conforme apresentado na Figura 6 abaixo:

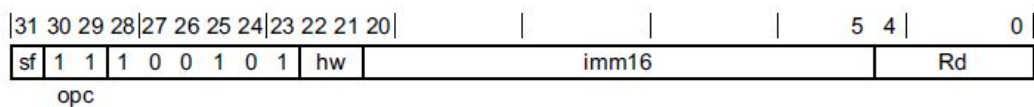


Figura 6: Formato das instruções MOV.

Para essa instrução, o conteúdo do registrador destino Rd é colocado na primeira entrada (A) e o valor da instrução estendido, com os bits significativos (de 22 a 5 intactos). A definição desta instrução é que o campo hw seleciona o “offset” do Mov, adicionando o valor imediato de 16 bits aos respectivos bits do registrador destino segundo, conforme explicitado na Tabela 2 a seguir.

hw	Descrição
00	Altera bits de 0 a 15
01	Altera bits de 16 a 31
10	Altera bits de 32 a 47
11	Altera bits de 48 a 63

Tabela 2: Bits a serem alterados conforme escolha da flag hw de uma instrução MOV.

Para a instrução MovK, os demais bits são mantidos. Já para a instrução MovZ, os demais bits são substituídos por zero. Para essas instruções, a solução mais direta é a utilização de um Mux 4x1, em que o valor de hw seleciona a saída.

A entrada do mux, para a instrução de MovZ é:

```
with movamount select movz_result <=
  '0' & x"000000000000" & b_signal(15 downto 0) when "00", --zero shift
  '0' & x"00000000" & b_signal(15 downto 0) & x"0000" when "01", --16
shift
  '0' & x"0000" & b_signal(15 downto 0) & x"00000000" when "10", --32
shift
  '0' & b_signal(15 downto 0) & x"000000000000" when "11", --48 shift
  (others=>'0') when others;
```

Já para o MovK:

```
--movk result
with movamount select movk_result <=
  '0' & (a_signal(63 downto 16) & b_signal(15 downto 0)) when "00", --zero
shift
  '0' & a_signal(63 downto 32) & b_signal(15 downto 0) & a_signal(15 downto
0) when "01", --16 shift
  '0' & a_signal(63 downto 48) & b_signal(15 downto 0) & a_signal(31 downto
0)when "10", --32 shift
  '0' & b_signal(15 downto 0) & a_signal(47 downto 0) when "11", --48
shift
  (others=>'0') when others;
```

Assim, a ULA foi expandida de modo a suportar todas as instruções. Por fim, é descrita a decodificação realizada pela ALUControl, expandindo os dois bits de ALUOp da unidade de controle para o requerido ALUSel. Essa decodificação é resumida pela Tabela 3 apresentada:

Codificação ALUOp			
00			
Instruções	10 downto 5	5 downto 0	ALUSel
Diversas			0010
01			
Diversas			0110
10			
CBZ	101101	100XXX	0111
CBNZ	101101	101XXX	0111
AND	100010	1010000	0000
Add	100010	1011000	0010
Addi	100100	00100x	0010
ANDI	100100	01000x	0000
ADDIS	101100	00100X	0000
Inclusive OR I	101100	01000X	0011
Exclusive OR	110010	010000	0001
SUB	110010	011000	0110
11	10 downto 5	5 downto 0	
ANDS	111010	010000	0000
ANDIS	111100	010000	0000
MOVK	111100	0101XX	0100
ADDs	101010	011000	0010
Inclusive OR	101010	010000	0011
MOVZ	110100	0101XX	0101
LSR	110100	011010	1000
LSL	110100	011011	1001
EORI	110100	01000X	0001
SUBI	110100	00100X	0110

Tabela 3: Codificação de ALUSel conforme ALUOp selecionada.

Essa separação foi realizada seguindo indicações do apêndice do livro [3]. Essa divisão claramente não produziu um resultado bom no sentido da implementação em hardware, já que as ALUOp 10 e 11 codificam uma série de operações, enquanto 00 e 01 codificam, respectivamente, Soma e Subtração.

O livro não justificou essa escolha de design. Suspeita-se que essa codificação permite que algumas instruções tenham sua operação já resolvidas no estágio de decodificação, como para as instruções de Load/Store. Por fim, o trabalho da implementação em hardware dessa decodificação consiste em determinar, a partir da tabela anterior, quais os bits são necessários para realizar a decodificação da operação, para cada grupo, com menor número de bits.

2.4. Interface com unidade de Ponto Flutuante

Para interagir com a unidade de ponto flutuante, foi convencionado com o grupo dessa unidade que:

- Do ponto de vista do processador, essa unidade seria completamente assíncrona, recebendo instruções e subindo uma flag de done quando acabado.
- A unidade está contida no estágio de execute.
- A entrada da unidade de controle são dois registradores lidos e os 11 bits mais significantes da instrução, que codificam a operação, e o campo de shamt da instrução. A decodificação completa da instrução é feita dentro dessa unidade.

Assim, para atender a essa especificação, foi proposta a seguinte interface:

```
entity fp_unity is

    port (
        A,B : in bit_vector (63 downto 0); -- inputs
        op  : in bit_vector (10 downto 0); -- op for instr
        sh  : in bit_vector (5 downto 0);  -- shamt for instr
        O   : out bit_vector (63 downto 0);
        done : out bit                    -- can continue
    );
end fp_unity;
```

Desta maneira, para o processador, essa unidade funciona como uma instrução aritmética do tipo R normal, havendo apenas a necessidade de se selecionar o resultado dessa unidade ou da ULA principal. Com isso, foi necessário adicionar um novo sinal de controle: fp, que seleciona, no estágio de Writeback, se o resultado a ser escrito no registrador foi gerado pela ULA ou pela unidade de ponto flutuante.

Como o mecanismo para permitir a escrita da unidade de ponto flutuante de forma assíncrona no pipeline era mais elaborado, foi escolhido apenas atrasar a execução da próxima instrução até que a unidade de ponto flutuante escrevesse seu resultado.

Isso foi escolhido pois, como outros grupos ficaram encarregados de implementar esquemas de execução fora-de-ordem, esses atrasos seriam resolvidos dentro dessas unidades, gerando um grande speed-up de execução dessas instruções para esse processador.

2.5 Interface com o módulo de memória

Para interagir com os caches de memória, foram definidas duas interfaces, uma para o cache de dados e outra para o cache de instruções. Isso se deve ao fato do cache de instruções ser somente para leitura e o cache de dados para leituras e escritas, devendo suportar escritas de 8, 16, 32 e 64 bits. A interface para o cache de instruções é:

```
entity rom is
  generic (
    addressSize : natural := 64;
    wordSize    : natural := 32;
  );
  port (
    addr : in  bit_vector(addressSize-1 downto 0);
    data : out bit_vector(wordSize-1  downto 0);
    hit  : out bit
  );
end rom;
```

Em que uma palavra é lida a cada ciclo de execução. Já para o cache de dados:

```
entity ram is
  generic (
    addressSize : natural := 64;
    wordSize    : natural := 64;
    byteSize    : natural := 8
  );
  port (
    ck, wr : in  bit;
    addr   : in  bit_vector(addressSize-1 downto 0);
    data_i : in  bit_vector(wordSize-1  downto 0);
    data_o : out bit_vector(wordSize-1  downto 0);
    NumBytes : in bit_vector(1 downto 0);
    hit     : out bit
  );
end ram;
```


Ambos os caches possuem sinais de hit e caso, ao final do ciclo, ocorra um miss, o pipeline deve ser atrasado. Esse atraso foi implementado de modo simples, desligando-se os clock-enable dos registradores de estágio.

2.6. Projeto da Unidade de Controle Completa

O projeto completo da unidade de controle consistiu em transcrever a planilha de instruções/sinais de controle para um grande “case switch”, colocando os devidos sinais de controle. A tabela completa, reproduzida no Apêndice A, contém todas as instruções estudadas. O arquivo do planilhas do libreoffice encontra-se na pasta docs/armv8 instrucoes.ods

3. Dependências com outros grupos

Para a execução mínima do projeto, conforme apresentado no planejamento do projeto, detectam-se duas dependências principais:

- Hierarquia de memória (4), responsável por controlar o acesso a memória e por meio da estratégia de cache aumentar o desempenho do processador;
- Unidade de ponto flutuante (3), responsável pela execução de parte das instruções do LEGv8.

Além disso, visando uma execução mais eficiente do processador, seria interessante integrar com os projetos de execução fora de ordem, Scoreboard (10) ou Tomasulo (11) e previsão de desvios (9). Para escrever programas para a plataforma, poderia-se utilizar o compilador do grupo (1).

Contudo, todas essas dependências não precisam ser tratadas com urgência pois, uma vez definindo-se as interfaces com (4) e (3), o projeto pode ser desenvolvido sem dificuldades com *mockups*, de modo a facilitar a integração com os projetos dos demais grupos.

Contudo, para os grupos 10, 11 e 12, a arquitetura da solução desse projeto é indispensável. Assim, logo nas primeiras semanas de projeto, essas interfaces deveriam ser definidas. Conforme acordado com os demais grupos, a primeira proposta para as interfaces foi a primeira pendência a ser resolvida com os demais grupos. Depois disso, eventuais modificações foram acordadas nas reuniões semanais. Dessas dependências, as que foram resolvidas são:

- Execução especulativa, em que o grupo, ao final do semestre, conseguiu adaptar o processador para demonstrar a execução do projeto deles, tendo a nossa ajuda para acertar as interfaces.
- Grupo do boot do sistema operacional: embora fosse uma dependência não descrita inicialmente, por falta de conhecimento, precisou de uma descrição exata sobre quais as instruções disponíveis para que fosse possível efetuar o boot.
- Definiu-se a interface com a unidade de ponto flutuante.

As demais não aconteceram, já que não foi possível contatar os demais grupos.

4. Mockups

Para substituir a dependência com a unidade de ponto flutuante, foram feitas descrições comportamentais em alto nível de abstração da unidade, tornando possíveis simulações. Com isso, a partir das interfaces de comunicação definidas entre as unidades, algumas linhas de código são suficientes para adaptar o projeto aqui descrito com os demais. Como para o processador o resultado dessa unidade é independente, ela foi colocada como zero para qualquer instrução.

Já para a hierarquia de memória, a substituição é ainda mais simples. Dado que para os programas de teste apenas pequenas memórias são necessárias, utilizará-se duas memórias pequenas e rápidas, separadas, para dados e instrução. Com isso, é possível emular a comportamento da hierarquia de memória, sendo inclusive sintetizável.

Já as demais dependências são “*nice-to-have*”, sendo possível superá-las apenas com configurações:

- O compilador pode ser facilmente substituído pelo assembler AS do arm toolchains, já que os programas de teste são consideravelmente simples. Eventuais discrepâncias de ARM e LEG, como a codificação do registrador XZR e SP¹ podem ser alteradas diretamente no binário.
- Para a execução fora de ordem e previsão de desvio, basta adicionar NOP’S ao código de máquina, visando eliminar as dependências de dados e controle. A execução ficará mais lenta, mas será possível validar o hardware projetado.

Como a integração com todos os grupos não foi efetuada, todos esses mockups foram utilizados.

¹ Na arquitetura ARM, tanto o SP quanto o XZR são codificados pelo número 31. Já no LEGv8, o Registrador número 31 é sempre XZR (também para outros tipos de dados) e o SP é o registrador de número 28.

5. Resultados e Simulações

Para verificar o funcionamento do processador, foram elaborados diferentes tipos de testes, com o intuito de avaliar a corretude do código desenvolvido em diferentes etapas. Deste modo, foram realizados pequenos testes unitários para cada módulo por meio do emprego das cláusulas `assert` e `report` de VHDL, a exemplo do que é mostrado nas figuras 7 e 8, apresentadas abaixo.

```
when "111110" =>
  report "Entrei no 111110";
  if (Instruction(22) = '1') then
    --AluOp 00
    -- Load Register Unscaled offset == 11111000010
    report "load";
    Reg2Loc      <= '0';
    Uncondbranch <= '0';
    Branch       <= '0';
    MemRead      <= '1';
    MemtoReg     <= '1';
    ALUOp        <= "00";
    MemWrite     <= '0';
    ALUSrc       <= '1';
    RegWrite     <= '1';
    bcond        <= '0';
    setflags     <= '0';
    bregister    <= '0';
    blink        <= '0';
    zeroext0     <= '0';
    zeroext1     <= '0';
    zeroext2     <= '0';
    exclusive    <= '0';
    numBytes     <= "00";
    fp           <= '0';
```

Figura 7: Exemplo de código a ser testado: tratamento realizado pela unidade de controle para a instrução de load.

```
Instruction <= '10111000100'; -- load word unscaled register half
run 10 ns;
assert Reg2Loc = '0'      report "falha em LDUR: Reg2Loc inesperado" severity error;
assert Uncondbranch = '0' report "falha em LDUR: Uncondbranch inesperado" severity error;
assert Branch = '0'      report "falha em LDUR: Branch inesperado" severity error;
assert MemRead = '1'     report "falha em LDUR: MemRead inesperado" severity error;
assert MemtoReg = '1'    report "falha em LDUR: MemtoReg inesperado" severity error;
assert ALUOp = "00"      report "falha em LDUR: ALUOp inesperado" severity error;
assert MemWrite = '0'    report "falha em LDUR: MemWrite inesperado" severity error;
assert ALUSrc = '1'      report "falha em LDUR: ALUSrc inesperado" severity error;
assert RegWrite = '1'    report "falha em LDUR: RegWrite inesperado" severity error;
assert bcond = '0'       report "falha em LDUR: bcond inesperado" severity error;
assert setflags = '0'    report "falha em LDUR: setflags inesperado" severity error;
assert bregister = '0'   report "falha em LDUR: bregister inesperado" severity error;
assert blink = '0'       report "falha em LDUR: blink inesperado" severity error;
assert zeroext0 = '0'    report "falha em LDUR: zeroext0 inesperado" severity error;
assert zeroext1 = '0'    report "falha em LDUR: zeroext1 inesperado" severity error;
assert zeroext2 = '1'    report "falha em LDUR: zeroext2 inesperado" severity error;
assert exclusive = '0'   report "falha em LDUR: exclusive inesperado" severity error;
assert numBytes = "01"   report "falha em LDUR: numBytes inesperado" severity error;
```

Já para os testes sistêmicos, programas curtos foram diretamente codificados na memória ROM, a exemplo do que é mostrado na figura 9. Em seguida, os resultados dos mesmos foram avaliados diretamente a partir da análise das formas de onda, conforme as figuras 10.

(b)

Figura 9: sequências de instruções utilizadas para avaliação dos resultados; em (a), um trecho do código utilizado para avaliar as operações executadas pela ULA; em (b), um trecho de código utilizado para verificar os resultados produzidos.

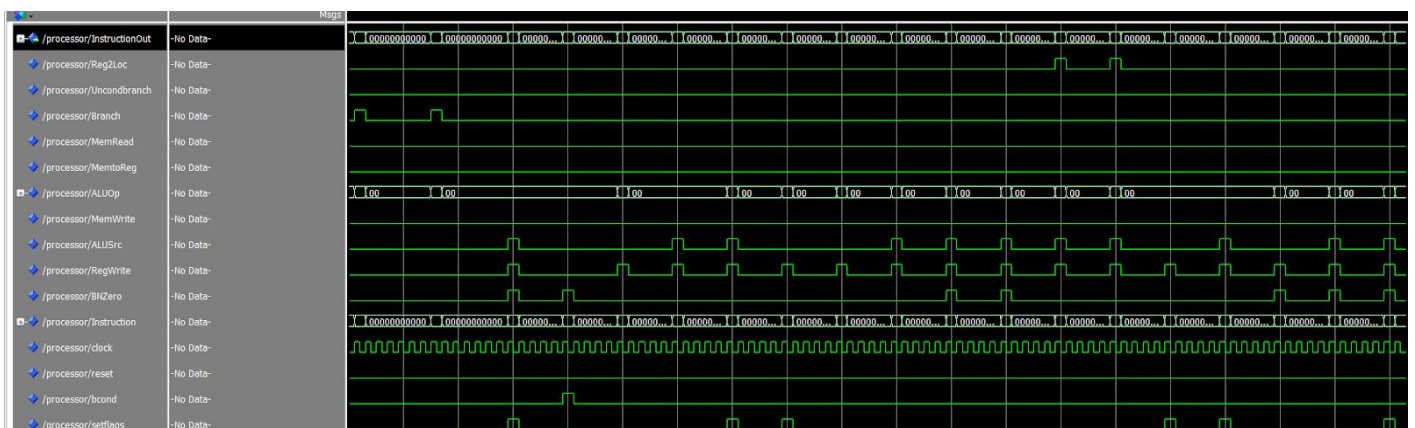


Figura 10: forma de onda resultante da simulação do processador a partir de código da ROM apresentado parcialmente na figura 9(a).

Por fim, para facilitar o setup de testes e a verificação de resultados, foi utilizada a interface de visualização e edição de conteúdo de memória do ModelSim, apresentada na figura 11. A partir da mesma, foi possível verificar mudanças de valores a cada iteração de execução das simulações feitas.

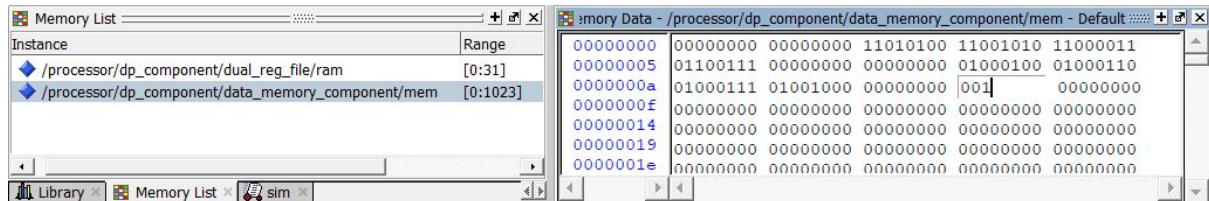


Figura 11: interface Memory List, utilizada para auxílio de verificação de resultados e setup de testes.

Por meio destas ferramentas, constatou-se o correto funcionamento do processador desenvolvido para a execução de programas simples.

6. Conclusões

Neste projeto, foi construído um processador básico, com todas as operações previstas e propostas. Foram mostradas as principais decisões de projeto envolvidas na construção do projeto, explicando o processo de decisão utilizado para fazer as expansões, que permite, de forma análoga, adicionar novas funcionalidades a outros projetos de hardware.

Com respeito a integração do trabalho com os demais grupos, não foi possível avançar muito nesse sentido, já que se organizar com outros grupos no período extra classe se mostrou um processo difícil. O estudo desse conjunto de instruções, do ponto de vista de arquiteto de hardware, permitiu uma visão melhor sobre o conjunto de instruções, com usos, facilidades e limitações, inclusive do ponto de vista da programação.

Referências

- [1] ARM® Architecture Reference Manual ARMv8, for ARMv8-A architecture profile
- [2] D. A. Patterson e J. L. Hennessy. Computer Organization and Design ARM Edition: The Hardware Software Interface. The Morgan Kaufmann Series in Computer Architecture and Design. Elsevier Science, 2016.
- [3] Gajski, D. D. e Abdi, S. e Gerstlauer, A. e Schirner, G., Embedded System Design, Springer, 2009