

UNIVERSIDADE FEDERAL DE SANTA MARIA
CENTRO DE TECNOLOGIA
CURSO DE GRADUAÇÃO EM ENGENHARIA ELÉTRICA

Kevin Paula Morais

**RELATÓRIO DE ESTÁGIO: DESENVOLVIMENTO DE UM
PROCESSADOR RISC-V RV32I**

Santa Maria, RS
2019

LISTA DE FIGURAS

Figura 2.1 – Codificação do tamanho da instrução.	11
Figura 2.2 – Caminho de dados.	17
Figura 2.3 – Demonstração da perda de performance pela latência da memória.	19
Figura 2.4 – Unidade de predição de desvios condicionais.	20
Figura 2.5 – Visão de topo dos elementos que constituem a memória do sistema, um decodificador que controla o acesso a memória de instruções (ROM) e a de dados (RAM).	21
Figura 2.6 – Detecção de instruções RVC no momento de leitura da ROM.	21
Figura 2.7 – Mapeamento da memória.	22
Figura 2.8 – Registrador de controle da UART, R_ND e TX_R indicam recepção e transmissão pronta, respectivamente.	23
Figura 2.9 – Hierarquia do sistema de memória.	24
Figura 2.10 – Implementação de dois níveis da memória cache.	25
Figura 2.11 – Mapeamento direto para uma cache de 1 kB.	26
Figura 2.12 – Mapeamento associativo por conjunto 4-way, 1 kB.	28
Figura 2.13 – Conexões da FPGA Spartan-6 com a Cellular RAM na placa Nexys3. .	31
Figura 2.14 – Diagrama de tempo da Cellular RAM no modo assíncrono durante uma operação de escrita, controlada por MT-CE, designado como CE# no diagrama.	33
Figura 2.15 – Diagrama de tempo da Cellular RAM no modo assíncrono page.	34
Figura 3.1 – Visão geral do processador.	35
Figura 3.2 – Formato de instruções Register-Register.	36
Figura 3.3 – Formato de instruções Register-Immediate.	37
Figura 3.4 – Formato das instruções LUI e AUIPC.	38
Figura 3.5 – Formato de instruções de desvio incondicional.	38
Figura 3.6 – Formato de instruções de desvio condicional.	39
Figura 3.7 – Formato de instruções Load/Store.	40
Figura 3.8 – Formato de instruções CSR e MRET.	40
Figura 3.9 – Mapeamento da memória implementado no script de link.	43
Figura 3.10 – Fluxo de tratamento de exceções/interrupções.	46
Figura 3.11 – Caminho de dados do núcleo, estão ocultadas as unidades de forward, e detecção de load-stall do estágio ID.	48
Figura 3.12 – Diagrama de tempo da busca de instruções.	50
Figura 3.13 – Detecção de exceções/interrupções.	52
Figura 3.14 – Execução de instruções CSR.	55
Figura 3.15 – Lógica de reset do núcleo.	58
Figura 3.16 – Diagrama de tempo da operação de reset do núcleo.	59
Figura 3.17 – Banco de registradores.	60
Figura 3.18 – Lógica interna da ALU, com as seções S1, S2, S3 e S4.	61
Figura 3.19 – Sistema preditor de desvios dinâmico.	64
Figura 3.20 – Diagrama de um flush no núcleo, afetando a cache de instrução.	68
Figura 3.21 – Máquina de estados finitos da cache de dados.	69
Figura 3.22 – Máquina de estados finitos responsável pelo controle de acesso da cache a memória principal.	71
Figura 3.23 – Portas de entrada e saída da interface com a memória.	72

Figura 3.24 – Máquina de estados finitos da interface com a memória RAM no modo assíncrono padrão, aonde os estados CR1, CR2, CR3 e P são ignorados.	74
Figura 3.25 – Máquina de estados finitos da interface com a memória para o modo page.	76
Figura 3.26 – Demonstração de acesso da página. A Figura (a) contém um acesso alinhado, que resulta no fim da página com palavra completa, e consequentemente não é necessária a leitura de uma palavra adicional da próxima página. Já no exemplo (b), ao terminar a página se tem uma palavra incompleta, se tornando necessário um acesso extra apenas para completa-lá.	78
Figura 3.27 – Controle interno do acesso aos dispositivos I/O.	80
Figura 3.28 – Registrador de controle da UART.	81
Figura 3.29 – Controle da UART.	82
Figura 4.1 – Entradas e saídas implementadas na FPGA.	86
Figura 4.2 – Caminho crítico do núcleo.	87
Figura 4.3 – Caminhos de propagação do banco de registradores.	88

LISTA DE ABREVIATURAS E SIGLAS

<i>BRAM</i>	Block RAM
<i>CI</i>	Circuito Integrado
<i>DMA</i>	Direct Memory Access
<i>DSP</i>	Digital Signal Processor
<i>FF</i>	Flip-Flop
<i>FIFO</i>	First-In First-Out
<i>FPGA</i>	Field-Programmable Gate Array
<i>HDL</i>	Hardware Description Language
<i>LIFO</i>	Least-In First-Out
<i>LFU</i>	Least Frequently Used
<i>LRU</i>	Least Recently Used
<i>LSB</i>	Least Significant Bit
<i>MSB</i>	Most Significant Bit
<i>RAM</i>	Random Access Memory
<i>PC</i>	Program Counter
<i>SO</i>	Sistema Operacional
<i>UART</i>	Universal Asynchronous Receiver/Transmitter
<i>USB</i>	Universal Serial Bus
<i>VHDL</i>	VHSIC Hardware Description Language
<i>VHSIC</i>	Very High Speed Integrated Circuits

SUMÁRIO

1	INTRODUÇÃO	7
1.1	OBJETIVO GERAL	7
1.2	OBJETIVOS ESPECÍFICOS	8
1.2.1	Núcleo	8
1.2.2	Memória	8
1.2.3	Interrupções e Exceções	9
1.2.4	ISA RISC-V	9
2	REVISÃO BIBLIOGRÁFICA	10
2.1	RISC-V	10
2.1.1	Base	10
2.1.2	Extensão C	12
2.1.3	Instruções CSR - Registradores de Controle/Status	12
2.1.4	Nível Privilegiado	13
2.2	PROCESSADOR RV32EC	14
2.2.1	Núcleo	16
2.2.1.1	<i>Dependências Estruturais e de Dados</i>	18
2.2.1.2	<i>Dependências de Controle</i>	18
2.2.2	Memória	20
2.2.3	Dispositivos I/O	22
2.3	SISTEMA DE MEMÓRIA	23
2.3.1	Cache	24
2.3.1.1	<i>Mapeamento</i>	25
2.3.2	Coerência de Dados	28
2.4	PLACA NEXYS3	29
2.4.1	FPGA Spartan-6	30
2.4.1.1	<i>Memória</i>	30
2.4.2	Cellular RAM	30
2.4.2.1	<i>Registradores de Controle</i>	32
2.4.2.2	<i>Modo Assíncrono</i>	32
2.4.2.3	<i>Modo Page</i>	32
3	METODOLOGIA E ESPECIFICAÇÕES	35
3.1	PROGRAMAÇÃO	36
3.1.1	Instruções	36
3.1.1.1	<i>Operação Register-Register</i>	36
3.1.1.2	<i>Operação Register-Immediate</i>	37
3.1.1.3	<i>Desvios Incondicionais</i>	38
3.1.1.4	<i>Desvios Condicionais</i>	39
3.1.1.5	<i>Load e Store</i>	39
3.1.1.6	<i>Instruções de Controle</i>	40
3.1.2	Compilação em C/ASM	41
3.1.3	Mapeamento da Memória	42
3.1.3.1	<i>Seção Bootloader</i>	44
3.1.3.2	<i>Seção m_excp/intr_vector</i>	45
3.1.3.3	<i>Seção m_trap_handler</i>	45
3.1.3.4	<i>Seção text e sdata</i>	46

3.2	NÚCLEO.....	46
3.2.1	Busca de Instrução	47
3.2.2	Busca de Dados	50
3.2.3	Detecção de Exceção/Interrupção.....	51
3.2.4	Registradores de Controle/Status	53
3.2.4.1	<i>mstatus</i>	55
3.2.4.2	<i>mie</i>	56
3.2.4.3	<i>mtvec</i>	56
3.2.4.4	<i>mepc</i>	56
3.2.4.5	<i>mcause</i>	57
3.2.5	Lógica de Reset.....	57
3.2.6	Banco de Registradores	58
3.2.7	Unidade Lógica e Aritmética	60
3.2.8	Predito de Desvios	62
3.3	MEMÓRIA CACHE.....	65
3.3.1	Coerência de Dados	66
3.3.2	Lógica de Reset.....	66
3.3.3	Inicialização da Cache	67
3.3.4	Cache de Instrução	67
3.3.5	Cache de Dados	68
3.3.6	Controle da Cache	70
3.4	CONTROLADOR PARA CELLULAR RAM	71
3.4.1	Máquina de Estados Finitos	73
3.4.2	Modo Assíncrono	74
3.4.2.1	<i>Escrita</i>	74
3.4.2.2	<i>Leitura</i>	75
3.4.3	Modo Page	76
3.5	DISPOSITIVOS I/O	77
3.5.1	Controle	79
3.5.2	UART	81
4	RESULTADOS E DISCUSSÃO	84
4.1	SÍNTESE E IMPLEMENTAÇÃO	84
4.1.1	Análise I/O	85
4.1.2	Análise de Caminho Crítico do Núcleo	86
4.2	PROGRAMA TESTE	87
5	CONCLUSÃO	90
5.0.1	Sugestões para Trabalhos Futuros	91
5.0.1.1	<i>Núcleo</i>	91
5.0.1.2	<i>Memória Cache</i>	92
5.0.1.3	<i>Seção I/O</i>	92
	REFERÊNCIAS BIBLIOGRÁFICAS	93
	APÊNDICE A – DADOS DE SÍNTESE E IMPLEMENTAÇÃO	94
	APÊNDICE B – PROGRAMA TESTE	98

1 INTRODUÇÃO

Atualmente diversas ISAs são implementadas em diferentes hardwares, como a do processador MIPS (PATTERSON; HENNESSY, 2004), ou então os processadores encontrados em sistemas embarcados contando com a arquitetura ARM. Sua influência no hardware final do sistema que a executa afeta diretamente a performance final do dispositivo. Com esse fator influenciador, a ISA do RISC-V surge como uma proposta para conjuntos de instruções, com livre implementação comercial e acadêmica. Seu projeto se iniciou na Universidade da Califórnia, em Berkeley, 2010.

O principal objetivo da ISA RISC-V consistia inicialmente em propósitos educacionais. Porém, de acordo com o crescimento do projeto, veio a se tornar uma opção de implementação em hardwares atuais. Sendo uma alternativa no requisito de performance relacionado a ISA, pois atualmente muitas das implementações são presas a questões industriais. Limitando-se as otimizações ao hardware projetado em torno da ISA selecionada. Como o nome deixa explícito, o RISC-V é formado por instruções do tipo RISC, sendo sua aplicação a base de extensões. Suas diversas instruções são separadas em vários tipos de extensões, que podem ou não ser implementadas no circuito, de acordo com o projetista. Garantindo assim flexibilidade em relação ao que será executado.

Este trabalho consiste na continuação do desenvolvimento de um processador de 32 bits, utilizando a ISA do RISC-V. Obteve-se anteriormente um núcleo de 32 bits RV32EC (base E e extensão C), com cinco estágios de pipeline. Também projetou-se uma memória simples para a execução de programas teste, e uma UART para comunicação com dispositivos externos. Nesta etapa do projeto se continua com um núcleo composto por cinco estágios de pipeline, porém com arquitetura modificada para RV32I. Projeteu-se também uma memória cache L1 separada em instruções e dados, e dois dispositivos periféricos, uma UART e um temporizador. Sua implementação é realizada na FPGA Spartan-6, sobre a placa de desenvolvimento Nexys-3. A descrição do hardware foi criada toda em VHDL.

1.1 OBJETIVO GERAL

Este trabalho tem como objetivo geral a continuação do desenvolvimento do processador RISC-V de 32 bits, descrito em VHDL e implementado na FPGA Spartan-6. Com o núcleo já contendo uma versão inicial finalizada, o foco principal é dado aos demais componentes essenciais ao processador, como a memória cache, dispositivos E/S, e também a parte de software.

1.2 OBJETIVOS ESPECÍFICOS

As subseções a seguir descrevem os objetivos relativos a cada componente do processador, e também de software.

1.2.1 Núcleo

Em relação ao núcleo, se tem como objetivo alterações a fim de mudar seu foco, que antes era para a redução de hardware com uso da arquitetura RV32EC. As modificações desejadas são:

- Remoção do suporte as instruções compressas de 16 bits da extensão C;
- Alteração da base RV32E para RV32I, aumentando o banco de registradores de 16 para 32;
 - Otimização do pipeline na busca de instruções e dados, removendo então a latência de leitura das BRAMs (Block RAMs);
 - Separação do barramento único para dados e instruções em dois, evitando o travamento do pipeline por disputa do barramento em diferentes estágios do pipeline;
 - Suporte a execução de instruções ainda não implementadas da base, como as de acesso a registradores de controle.

1.2.2 Memória

A memória utilizada no sistema consiste em uma descrição simples que utiliza BRAMs da FPGA Spartan-6, com suporte a acessos específicos como leitura apenas de byte ou meia palavra. Se tem por objetivo adaptá-la em duas memória cache L1, uma de instrução e outra de dados, com as seguintes características:

- Mapeamento do tipo direto, no qual irá ser estudada a viabilidade de implementar 1 a 2 duas palavras por linha, ou então, o uso do mapeamento associativo por conjunto;
- Obter no mínimo 16 kByte de cache de instrução, e 16 kByte de dados, através das BRAMs;
- Projeto de uma interface de controle da memória, a fim de utilizar dispositivos de memória externo a FPGA Spartan-6, tendo como alvo o CI disponível na placa de desenvolvimento Nexys3 utilizada no projeto.

1.2.3 Interrupções e Exceções

O uso de dispositivos I/O é um dos grande responsáveis por perda de desempenho no núcleo, por conta do uso da técnica de polling. Sendo assim se tem como objetivo a implementação do tratamento de interrupções e exceções no núcleo. Adaptando então os dispositivos periféricos presentes, a UART e o temporizador.

1.2.4 ISA RISC-V

Alterando o foco do processador para um de uso mais geral, se deseja agora o uso de instruções não implementadas ainda da base do RISC-V, como as de acesso a registradores de controle do núcleo. Sendo assim será utilizado também o manual de nível privilegiado de instruções (WATERMAN; ASANOVIĆ, 2017b), em que se descreve as características de três níveis de privilégio para implementação. O nível de Máquina (M-mode) é o único mandatório em um sistema, logo é o alvo de implementação.

2 REVISÃO BIBLIOGRÁFICA

Este capítulo apresenta uma revisão sobre a área do projeto, incluíndo conceitos teóricos, características técnicas de dispositivo utilizados, e dos resultados anteriormente obtidos do processador. Na Seção 2.1 se apresenta a ISA utilizada no projeto, o RISC-V. Na Seção 2.2 verifica-se o estado em que o processador encontra-se desde sua última avaliação. Na Seção 2.3 é realizada uma revisão teórica sobre a memória cache, e diferentes formas de implementa-lá. Na Seção 2.4 se apresenta o ambiente de desenvolvimento do projeto, a placa Nexys3, e seus principais componentes abordados nessa etapa da pesquisa.

2.1 RISC-V

O RISC-V consiste em uma ISA do tipo RISC (Reduced Instruction Set Complexity) que se caracteriza pelo livre uso em implementações tanto comerciais, quanto de pesquisa. Seu uso é dividido por base, sendo quatro atualmente, RV32I, RV64I, RV128I e RV32E, em que se deixa claro em cada uma o tamanho dos dados manipulados, 32 bits para a base RV32I, por exemplo. A base RV32E consiste na mesma que a RV32I, porém com alterações no acesso ao banco de registradores por meio das instruções. A Tabela 2.1 apresenta as bases e extensões da ISA, aonde as extensões são conjuntos de instruções extras que podem ser implementadas para expandir o funcionamento da arquitetura. Quando se implementa uma arquitetura com a base RV64I por exemplo, e as extensões A e F, a nomenclatura utilizada define primeiro a base e em seguida as extensões, ficando RV64IAF (WATERMAN; ASANOVIĆ, 2017a).

2.1.1 Base

A base RV32I é caracterizada por utilizar dados e instruções de 32 bits, contendo instruções unicamente para manipulação de dados do tipo inteiro. Sua implementação define um banco de registradores 32x32 bits a ser utilizado pelas instruções, e um registrador extra para endereçamento da memória, o PC (Program Counter). O banco de registradores é o mesmo a base 64I e 128I, variando apenas o tamanho do dado, ou seja, sendo 32x64 bits no 64I. Já para a base E se tem a alteração no tamanho do banco de 16 para 32 registradores de uso geral, visando a redução do hardware, ideal para sistemas embarcados em que se deseja obter o mínimo possível de uso da área do chip.

Tabela 2.1 – Tabela de bases e extensões do RISC-V

Base	Extensão
RV32I	M - Integer Multiplication and Division
RV64I	A - Atomic
RV128I	F - Single-Precision Floating-point
RV32E	D - Double-Precision Floating-point Q - Quad-Precision Floating-point L - Decimal Floating-point C - 16-bits Compressed Instructions B - Bit Manipulation J - Dynamic Languages T - Transactional Memory P - Packed-SIMD Extensions V - Vector Extensions N - User-Level Interrupts

Fonte: Autor.

Para se diferenciar o tamanho da instrução utilizada, são verificados seus LSb, conforme se demonstra na Figura 2.1. Quando os 2 LSb são diferentes de 11, temos uma instrução compressão da extensão C, de 16 bits. Nem todas as bases e extensões são compatíveis entre si para implementação, a E por exemplo, pode ser utilizada com as extensões M, F e C.

Figura 2.1 – Codificação do tamanho da instrução.

xxxxxxxxxxxxxxxxaa	16-bit ($aa \neq 11$)
xxxxxxxxxxxxxxxxxx xxxxxxxxxxxxbbb11	32-bit ($bbb \neq 111$)
...xxxx xxxxxxxxxxxxxxxxxx xxxxxxxxxx011111	48-bit
...xxxx xxxxxxxxxxxxxxxxxx xxxxxxxxxx0111111	64-bit
...xxxx xxxxxxxxxxxxxxxxxx xnnnxxxxx1111111	(80+16*nnn)-bit, $nnn \neq 111$
...xxxx xxxxxxxxxxxxxxxxxx x111xxxxx1111111	Reserved for ≥ 192 -bits

Byte Address: base+4 base+2 base

Fonte: Waterman e Asanović (2017a, p. 6).

2.1.2 Extensão C

A extensão C é caracterizada por disponibilizar instruções compressas de 16 bits, sendo cada uma projetada para ser possível de se expandir na sua correspondente da base. Quando se tem a implementação dessa extensão, o endereço de acesso a memória de instruções sofre um relaxamento, aonde agora não é mais necessário que cada instr. esteja alinhada de 4 em 4 bytes. Uma implementação prática da extensão C no hardware iria ser com o uso de um decodificador do tamanho da instr. na etapa de busca, enviando para o núcleo uma instrução equivalente da base. Assim o núcleo não precisa de decodificadores adicionais para os formatos de 16 bits. A detecção de seu tamanho é feita pelos 2 LSBs da instrução, que quando diferente de 11, indica que é da extensão C. Ressaltando que o valor do PC deve ser incrementado em 2 e não 4 quando se tem uma instrução RVC (nomenclatura das instruções compressas), pois agora o endereço está relaxado em relação ao alinhamento, e instruções de 32 bits podem ser alocadas em endereços desalinhados (2 em 2 bytes).

2.1.3 Instruções CSR - Registradores de Controle/Status

Os registradores de controle/status definidos pelo RISC-V, utilizados para monitoramento do hardware de acordo com o nível de privilégio discutido na Seção 2.1.4, podem ser acessados por meio de instruções CSR. Um total de 6 instruções estão definidas na base do RISC-V:

- **CSRRW**: realiza a leitura do valor antigo contido no registrador CSR e o armazena no banco de regs., enquanto se escreve o novo, resultando em uma troca de valor entre um registrador do banco e um regs. de controle;
- **CSRRS**: utiliza o dado de entrada como uma máscara na qual para cada bit que estiver em nível 1, sua posição equivalente do regs. de controle será colocado em nível 1, mantendo os demais inalterados, também se lê o valor antigo do regs. de controle e o salva no banco;
- **CSRRC**: utiliza o dado de entrada como uma máscara na qual para cada bit que estiver em nível 1, sua posição equivalente do regs. de controle será colocado em nível 0, mantendo os demais inalterados, também se lê o valor antigo do regs. de controle e o salva no banco;

As instruções CSRRWI, CSRRSI e CSRRCI funcionam da mesma forma que CSRRW, CSRRS e CSRRC respectivamente, porém o dado de entrada a ser escrito no regs. de controle não é mais provido pela saída do banco de regs.. Corresponde agora ao campo rs1 de endereçamento do banco, porém extendido de 5 bits para 32 (com 0s). Dessa

forma, apenas os 5 LSb podem ser escritos por meio dessas instruções, sendo útil para colocar em nível 1 alguma flag rapidamente, desde que esteja nos 5 LSb. O registrador mstatus por exemplo, possui o sinal de habilitação global de interrupção na posição 3 para nível de privilégio M (Machine).

2.1.4 Nível Privilegiado

O RISC-V define três níveis de privilégio durante a execução de programas no núcleo, M (Machine), S (Supervisor) e U (User), sendo atribuído registradores de controle próprios para cada um (WATERMAN; ASANOVIC, 2017b). Na implementação de nível privilegiado, o único mandatório é o de máquina (M), que geralmente possui acesso a todo o hardware, sendo utilizados os modos S e U para a execução de programas de usuário e/ou do sistema operacional. A Tabela 2.2 demonstra a possível aplicação do hardware, de acordo com os níveis de privilégio que ele suporta. Os registradores de nível M possuem diversos campos com informações a respeito de outros níveis de privilégio, que podem ser em geral conectados ao GND, em implementações simples que não utilizam mais que o nível M.

Tabela 2.2 – Intenções de uso dos níveis de privilégio em uma arquitetura RISC-V.

Número de Níveis	Modos Suportados	Uso
1	M	Sistema embarcado simples
2	M , U	Sistema embarcado seguro
3	M , S ,U	Sistemas operacionais como Unix

Fonte: Waterman e Asanović (2017b, p. 4).

O estado do hardware durante um nível de privilégio é acessado por meio de instruções CSR, lendo/escrevendo diretamente nos registradores de controle quando desejado, sendo possível ter um registrador de controle para cada nível. Todos os CSR possuem como prefixo o seu nível de privilégio, por exemplo, o registrador status para cada modo corresponde a mstatus (modo M), sstatus (modo S) e ustatus (modo U). Alguns dos registradores essenciais em um sistema que se deseja receber exceções/interrupções, com o nível M de privilégio, são:

- **mstatus**: contém o estado do hardware, como por exemplo, flags de habilitação global de exceções/interrupções;
- **medeleg**: atribuí a execução de uma determinada exceção a outro níveis de privilégio, quando desejado, fixo no GND quando se tem apenas o nível M;
- **mideleg**: atribuí a execução de uma determinada interrupção a outro níveis de privilégio, quando desejado, fixo no GND quando se tem apenas o nível M;

- **mie**: registrador de habilitação de exceção/interrupção, sendo separados em tipos, como exceções por software, interrupção por temporizadores, e interrupções externas;
- **mip**: registrador de interrupções pendentes, sendo complementar ao registrador mie;
- **mtvec**: contém o endereço base de desvio do PC para tratamento da trap, na ocorrência de uma exceção/interrupção, seus 2 LSb definem o modo de operação entre DIRECT e VECTOR, e não fazem parte do endereço;
- **mepc**: armazena o endereço da instrução que causou a exceção, que possuí os 2 LSb fixos em 0, pois o PC de retorno não pode estar desalinhado;
- **mcause**: contém um valor codificado que representa a causa da exceção/interrupção, a fim de auxiliar no software de tratamento;
- **mtval**: armazena endereços de desvio ou load/store, acesso a memória em geral, que são inválidos (desalinhados caso não se deseje).

Tanto mepc quanto mtvec devem armazenar endereços alinhados, com os 2 LSb em 0, do contrário, seriam a causa de outra exceção. No caso do mtvec, se utiliza os 2 LSb como 0 apenas para formar o endereço, o CSR de fato armazena uma codificação que indica o tipo de desvio realizado na recepção da trap. Quando 00, temos o modo DIRECT, no qual o PC desvia para o exato endereço contido em mtvec. Já para 01, modo VECTOR, se desvia o PC para o endereço de mtvec somado a codificação da causa na recepção de interrupções, para exceções, se realiza apenas o desvio para o endereço contido em mtvec.

As possíveis cause de traps são codificadas conforme a Tabela 2.3, com os valores atribuídos ao registrador mcause na recepção do evento. Interrupt corresponde ao MSb do registrador, que quando 1 indica uma interrupção, do contrário uma exceção. O código da exceção representa os 31 bits restantes.

2.2 PROCESSADOR RV32EC

A implementação da etapa anterior do projeto utilizou da base E e extensão C do RISC-V, revisados na Seção 2.1, na qual se define então um núcleo capaz de executar instruções de 32 bits, e versões compressas de 16 bits. Contendo um banco de registradores com tamanho 16x32 bits. Utilizou-se de todas as instruções da base e extensão, com exceção das que envolviam o controle do hardware, como instruções CSR (control/status register), ou então uso de ponto flutuante. Ignorou-se também chamadas a sistemas supervisores como EBREAK e ECALL (MORAIS, 2018).

Com essas especificações da ISA, projetou-se um núcleo pipeline de 5 estágios com execução em ordem, um bloco de controle para dois dispositivos I/O, uma UART

Tabela 2.3 – Codificação da causa da trap, armazenado no CSR mcause.

Interrupção	Código de Exceção	Descrição
1	0	Interrupção de Software U
1	1	Interrupção de Software S
1	2	Reservado
1	3	Interrupção de Software M
1	4	Interrupção de Temporizador U
1	5	Interrupção de Temporizador S
1	6	Reservado
1	7	Interrupção de Temporizador M
1	8	Interrupção Externa U
1	9	Interrupção Externa S
1	10	Reservado
1	11	Interrupção Externa M
1	≥ 12	Reservado
0	0	End. de Instrução Desalinhado
0	1	Falta de Acesso a Instrução
0	2	Instrução Ilegal
0	3	Breakpoint
0	4	End. de Load Desalinhado
0	5	Falta de Acesso Load
0	6	End. de Store/AMO Desalinhado
0	7	Falta de Acesso Store/AMO
0	8	Chamada de Sistema do Modo U
0	9	Chamada de Sistema do Modo S
0	10	Reservado
0	11	Chamada de Sistema do Modo M
0	12	Falta de Página de Instrução
0	13	Falta de Página de Load
0	14	Reservado
0	15	Falta de Página de Store/AMO
0	≥ 16	Reservado

Fonte: Waterman e Asanović (2017b, p. 35).

para comunicação com o computador pessoal e um temporizador. E por fim, um bloco de memória, que era conectado ao núcleo pipeline por um único barramento de dados, não havendo separação entre dados e instruções. A Tabela 2.4 contém a utilização de recursos da síntese e implementação do processador com núcleo pipeline, na FPGA Spartan-6, que corresponde a principal ferramenta do projeto. Nela percebe-se que ainda há espaço para o desenvolvimento de diversas funcionalidades no processador, e que se atingiu uma velocidade mínima de 100 MHz, que era o desejado, correspondendo a frequência do cristal oscilador disponível na placa Nexys3.

Tabela 2.4 – Resumo de utilização de recursos da FPGA Spartan-6.

Recursos	Disponível	Usado	Utilização (%)
Número de Slice Registers	18.224	1.137	06%
Número de Slices LUTs	9.112	2.137	23%
Número de Slices Ocupados	2.278	782	34%
Blocos de RAM	32	18	56%
Frequência Máxima (MHz)		100,09	

Fonte: Autor.

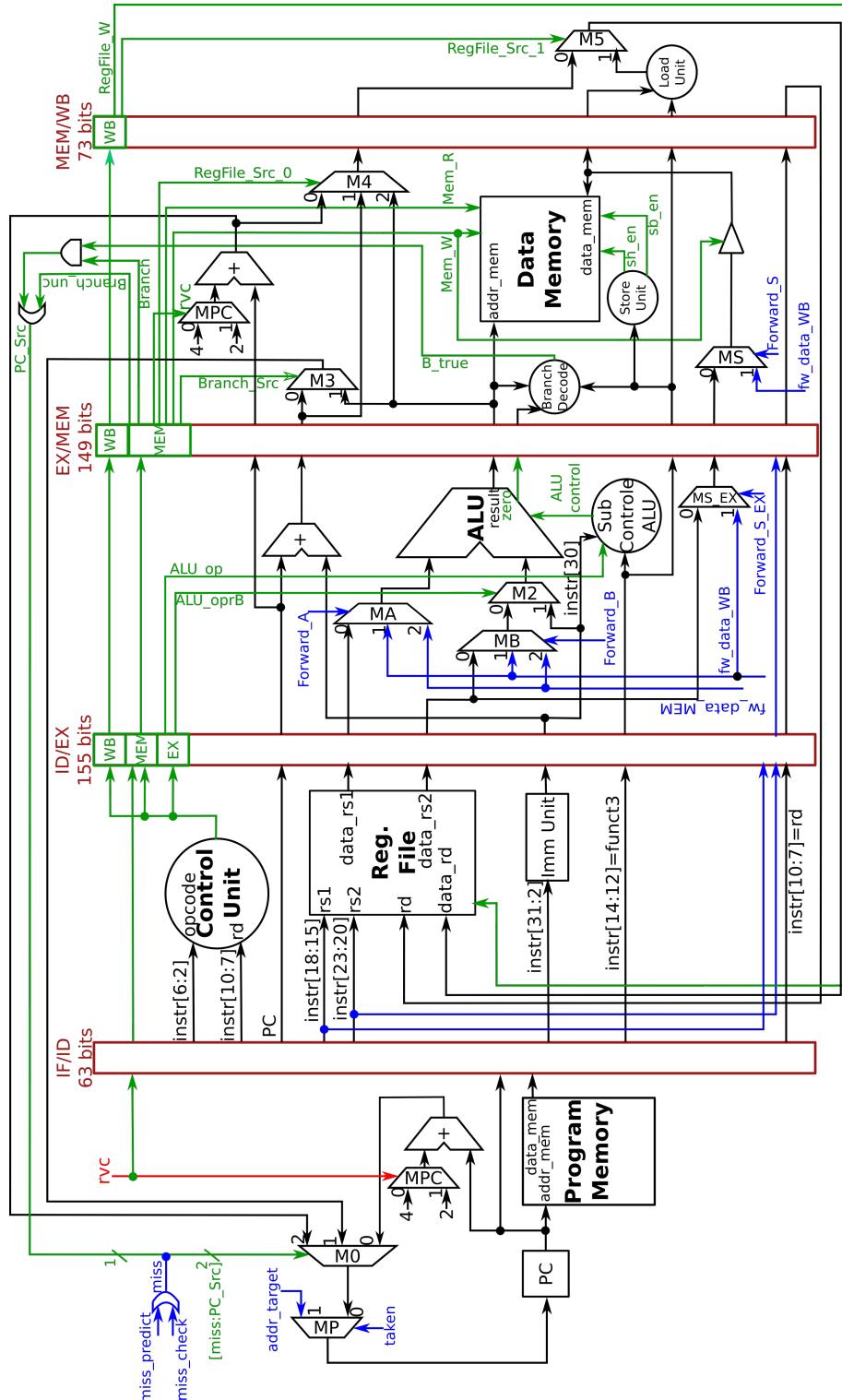
2.2.1 Núcleo

O núcleo foi projetado com um total de 5 estágios de pipeline, IF (busca de instrução), ID (decodificação da instrução), EX (execução da instrução), MEM (acesso a memória) e WB (escrita do resultado de volta), a Figura 2.2 apresenta o caminho de dados obtido, com as unidades de dependências (preditor e forward) ocultados. Apesar de demonstrar acesso a duas memórias, elas constituem uma única, logo se tem um barramento apenas para endereçamento de instr. e dados, sendo dividido entre o estágio IF e MEM. Esse barramento único define um dos principais causadores de perda de desempenho desse projeto, pois enquanto se acessava a memória no estágio MEM, era necessário gerar um stall no estágio IF, inserindo a instrução NOP no estágio ID. O hardware não possui lógica de interrupção/exceção, se tornando uma necessidade o uso de polling por software para acesso aos dispositivos I/O.

As instruções da extensão C eram decodificadas separadamente do núcleo, durante a etapa de busca na memória, sendo assim sua remoção/adição no hardware é extremamente simples. A única informação que o núcleo realmente precisa é de um bit indicando (rvc em nível 1) quando a instrução é RVC ou não. Isso é necessário para que se incremente o PC de acordo, para instruções RVC se utiliza um offset de 2 bytes, já para as demais da base E, 4 bytes. A unidade de Imm no estágio ID é responsável por decodificar o campo da instrução que forma o sinal imediato, ação necessário pelo fato da ISA do RISC-V embaralhar ao longo dos diferentes formatos de instruções os campos imediato. Porém para facilitar o hardware extensor, o MSb (a extensão é sempre pelo MSb) é sempre localizado no bit [31] da instrução.

O banco de registradores presente no estágio ID é um dos principais componentes do projeto, pois a base E define justamente a sua redução de 32 registradores de uso geral, para 16. Sua descrição em VHDL foi realizada de forma a utilizar LUTs, e não blocos de RAM. Desconsiderou-se o uso de BRAMs devido ao alto desperdício de uso das mesmas, que possuem uma capacidade de armazenamento muito superior a necessária por parte do banco. Outros fatores também influenciaram na decisão, como por exemplo, o fato de que BRAMs possuem apenas duas portas, e necessita-se de três, duas de leitura, e uma de escrita. Apesar de se utilizar LUTs, o uso de um reset geral no banco limita a capacidade

Figura 2.2 – Caminho de dados.



Fonte: Autor.

de implementação dessa pequena memória na FPGA Spartan-6, não se aproveitando dela como uma memória distribuída. Internamente o banco compara os registradores de leitura, e o destino de escrita, multiplexando para a saída o novo dado quando necessário (lógica de forward interna).

2.2.1.1 Dependências Estruturais e de Dados

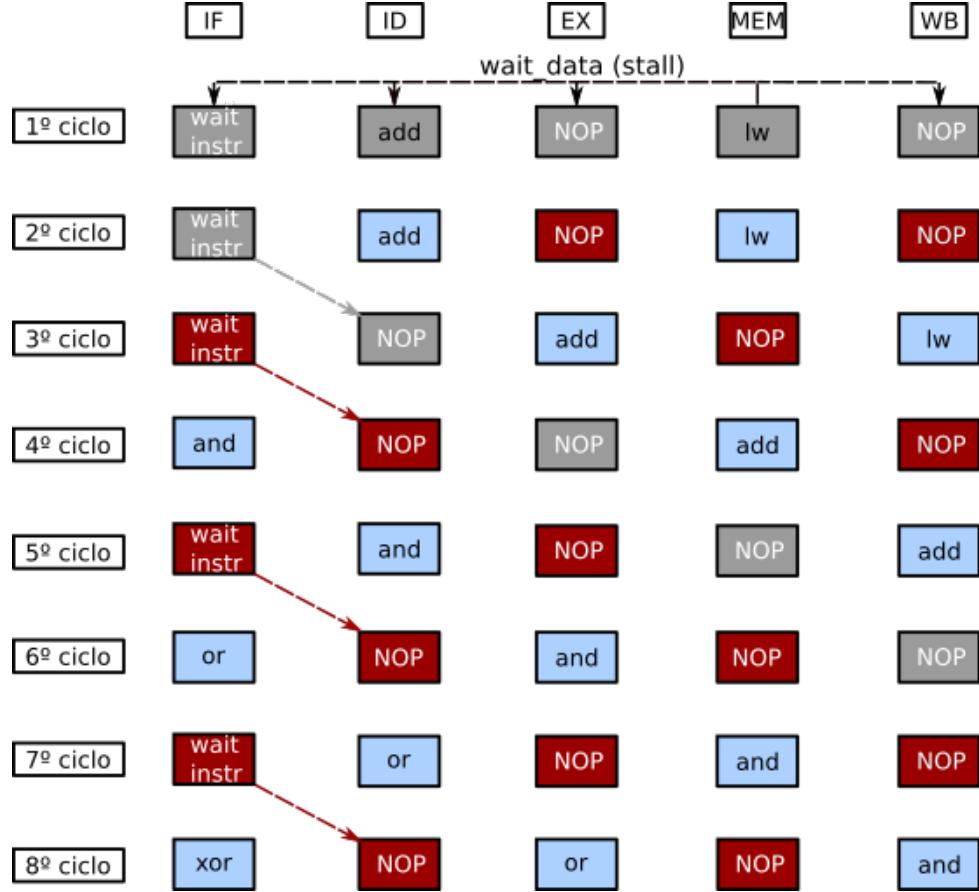
Outro fator que afeta o desempenho do caminho de dados obtido era o fato de que a memória, formada por blocos de RAM, possuía uma latência de 1 ciclo de acesso, sendo necessário inserir um NOP no pipeline a cada instrução buscada, somente com isso o desempenho já caia pela metade. O mesmo ciclo de espera acontece na busca de dados no estágio MEM. A Figura 2.3 demonstra um caso de grande perda de performance, em que no primeiro ciclo se tem uma instrução LW (load word) no estágio MEM, que por consequência trava o pipeline todo por um ciclo. Após retomar a execução no segundo ciclo, percebe-se que ainda não se tem uma instrução no estágio IF, pois ele teve que ceder o barramento para a busca de dados no ciclo 1, e no 2 para a leitura do resultado. Sendo assim, somente no terceiro ciclo em que se começa o acesso para busca da instrução, e no quarto que se realiza a leitura do dado disponível no barramento. Em conclusão, considerando o LW desde o estágio IF teríamos um total de 11 ciclos de relógio para a finalização de apenas 3 instruções, quebrando o desempenho do pipeline. Destaca-se que após o terceiro ciclo, mesmo sem a disputa pelo barramento ainda se tem um NOP entre cada instrução devido a espera pela memória.

A técnica de forwarding foi utilizada para resolver dependências de dados verdadeiros, aonde se destaca o fato de que devido a inserção de um NOP a cada instrução buscada (devido a latência da memória), certos caminhos do forward nunca ocorrem, e o hardware se torna inutilizável. Para ser claro, todos os caminhos de forward que ocorrem entre dois estágios adjacentes (WB para MEM, e MEM para EX) não ocorrem. A mesma característica é perceptível na unidade para detecção de load-stall, alocada no estágio ID. Sua função é inserir um NOP entre instruções LOAD e qualquer uma que utilize seu resultado, pois o forward não cobre essa dependência. Porém, o NOP já é inserido automaticamente no estágio IF, sendo desperdiçada a unidade de detecção.

2.2.1.2 Dependências de Controle

As dependências de controle, que se referem aos desvios condicionais/incondicionais, utilizam de um preditor de desvios dinâmicos, com lógica de predição por meio de uma máquina de estado de 2 bits, analisada e citada por Arora, Kotecha e Samyal (2013)

Figura 2.3 – Demonstração da perda de performance pela latência da memória.



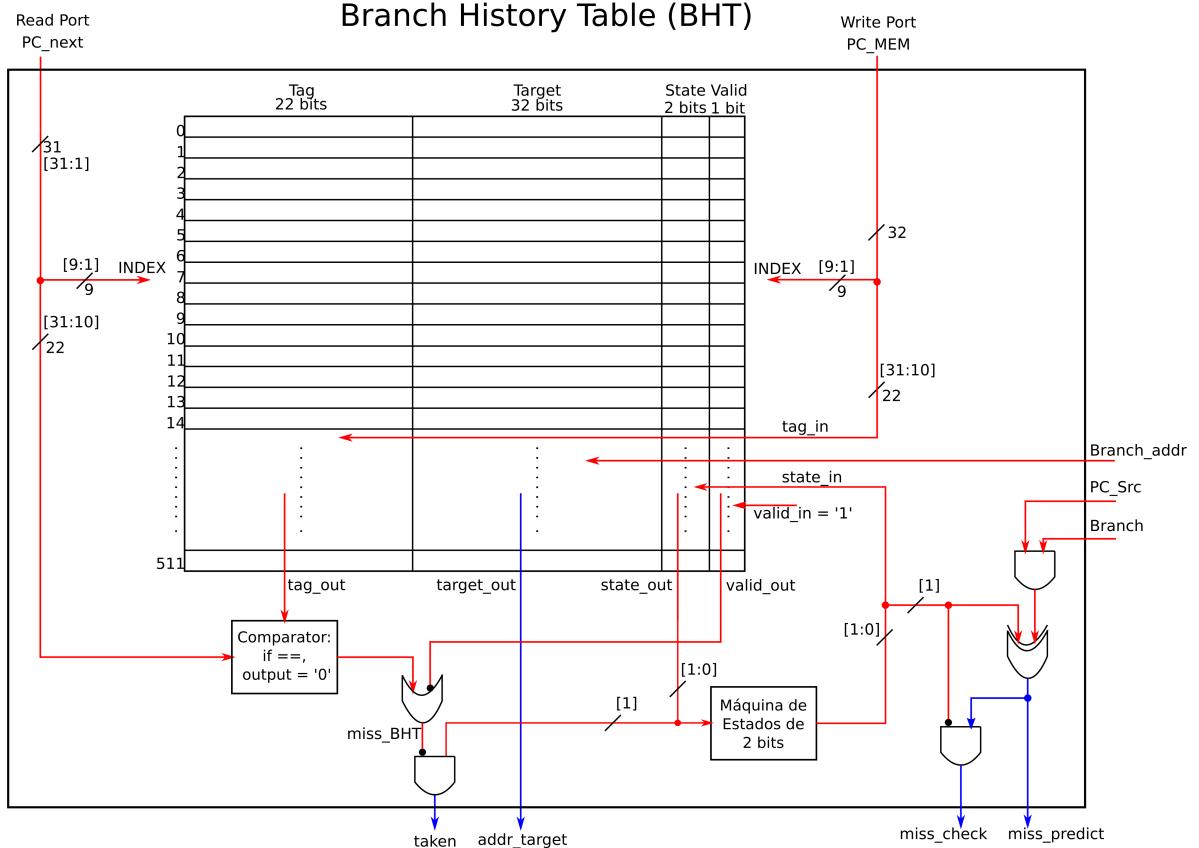
Fonte: Autor.

como um contador de saturação. O preditor entra em ação unicamente para instruções de desvio condicional (branch), por consequência, todos os desvios incondicionais (jump) irão causar o flush do pipeline. Optou-se por não utilizar da capacidade do preditor com desvios incondicionais devido sua menor ocorrência em comparação a branches, frequentemente utilizado em laços, e também pela capacidade de armazenamento utilizada na BHT (Branch History Table).

A BHT foi implementada por meio de dois BRAMs, um dedicado para armazenamento do endereço alvo da predição, e outro BRAM contendo a tag, bit de validade, e 2 bits de estado do branch contido naquela linha acessada. Com as possíveis configurações dos blocos de RAM da FPGA Spartan-6, obteve-se então um preditor de 512 linhas, com mapeamento direto, logo, sem algoritmo de substituição. O diagrama de blocos da Figura 2.4 contém o preditor, incluíndo a porta de leitura, acessada no estágio IF para busca de um branch, e a porta de escrita, que é acessada no estágio MEM, para atualização/inserção de um branch na tabela. Os sinal `addr_target` corresponde ao endereço alvo de predição, e o `taken` a flag ativa em 1, indicando que deve ser utilizado o endereço fornecido (`addr_target`). As flags `miss_check` e `miss_predict` são complementares, indicando

quando o preditor errou ao verificar o resultado o estágio MEM, e sobre qual predição errou (taken ou no taken).

Figura 2.4 – Unidade de predição de desvios condicionais.



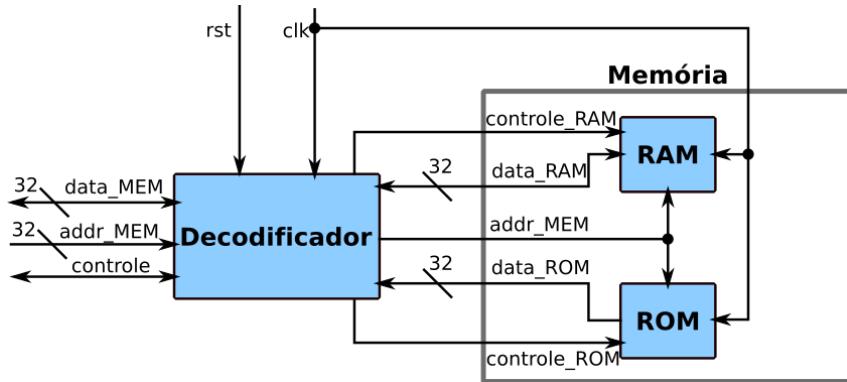
Fonte: Autor.

2.2.2 Memória

A memória foi implementada utilizando os blocos de RAM da FPGA, ocupando um total de 16 BRAMs, totalizando 32 kiB de memória. O componente foi projetado com um único barramento de endereço de dados e endereço, sendo visto pelo núcleo então como a memória principal do sistema. Internamente ela se dividia em dois blocos, um contendo as instruções, e outro dados, ambos de 16 kiB. A definição de qual estava sendo acessado era realizada por meio de um comparador entre o endereço acessado, e dois parâmetros que correspondem as regiões da memória que cada bloco utiliza. Na Figura 2.5 visualiza-se uma visão de topo da memória, com o decodificador definindo qual das memórias irá ter acesso ao barramento conectado ao núcleo, data_MEMORY e addr_MEMORY.

É de dentro da unidade de decodificação da memória que se origina o sinal de

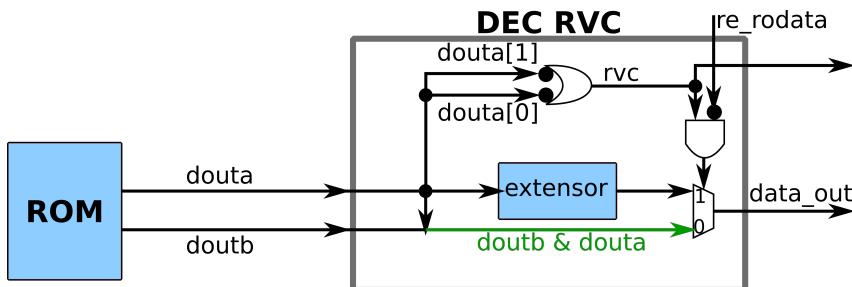
Figura 2.5 – Visão de topo dos elementos que constituem a memória do sistema, um decodificador que controla o acesso a memória de instruções (ROM) e a de dados (RAM).



Fonte: Autor.

entrada do núcleo, rvc, indicando quando a instrução é de 16 bits, e o PC deve ser incrementado por 2, e não 4. Sua lógica de detecção é verificada no bloco da Figura 2.6, em que se verifica os 2 LSb da instrução buscada na ROM. Quando for diferente de 11, corresponde a uma instrução RVC, e se multiplexa para a saída a instrução decodificada de 16 para sua correspondente de 32 bits, mantendo o núcleo sem a necessidade de qualquer lógica extra fora o incremento do PC. Também se envia a flag rvc para o núcleo. Caso os 2 MSb sejam 11, apenas se transfere para a saída os 32 bits originais da saída da ROM.

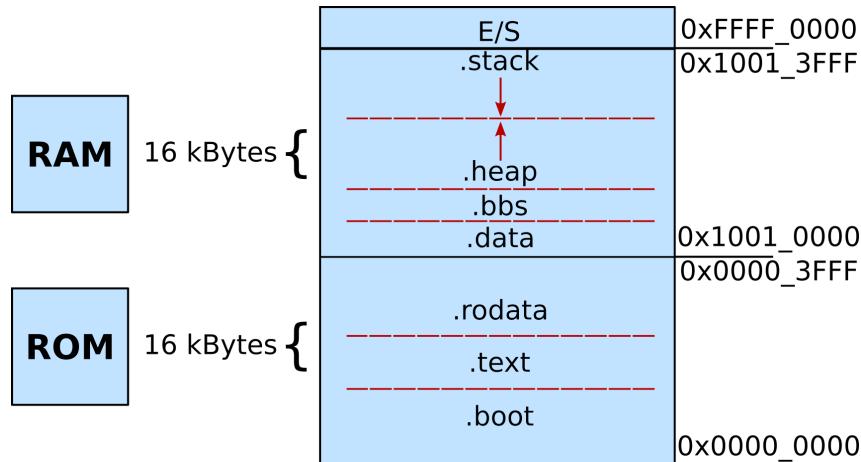
Figura 2.6 – Detecção de instruções RVC no momento de leitura da ROM.



Fonte: Autor.

As regiões foram divididas conforme o mapeamento de memória estabelecido na Figura 2.7, em que se aproveita de apenas 32 kiB de endereçamento, e não os 4 GiB possíveis com 32 bits. As seções de .boot, .text e .rodata (dados constantes) estão alocados no bloco da ROM (memória de instruções), enquanto o restante das seções essenciais, .data, .bss, .heap e .stack (crescendo em sentidos opostos) estão na RAM. O restante dos endereços até a seção I/O são ignorados, sendo desabilitado o acesso a memória pelo resultado do comparador. A seção I/O, como evidenciado, define um mapeamento direto na memória dos dispositivos. As regiões definidas foram implementadas em um script de link para compilação de programas em C/ASM.

Figura 2.7 – Mapeamento da memória.



Fonte: Autor.

2.2.3 Dispositivos I/O

Os dispositivos periféricos do núcleo se resumem a dois implementados para teste, uma UART para comunicação com o computador pessoal, e um temporizador de 32 bits. Nenhum deles gera interrupções, visto que o núcleo não possuí suporte a tal esquema. O temporizador funciona em sincronia com o sinal de clock do sistema, a 100 MHz com acesso tanto para leitura quanto para escrita. A Tabela 2.5 contém o mapeamento dos dispositivos na seção I/O da memória. Como consequência do mapeamento direto e da ISA do RISC-V não conter instruções específicas para acesso a I/O, a execução de LW e SW é utilizada para acesso aos dispositivos. Os registradores da UART ocupam apenas 8 bits dos 32 alocados, porém optou-se por manter assim para que o seu acesso ficasse alinhado em 4 bytes com operações LW e SW.

Tabela 2.5 – Endereços da seção E/S ocupados pela UART e pelo temporizador, indicando-se quais operações de acesso estão disponíveis.

Dispositivo E/S	Endereço	Função
UART Registrador de Controle	0xFFFF_0000	Leitura
UART Registrador de Dados	0xFFFF_0004	Leitura/Escrita
Temporizador	0xFFFF_0008	Leitura/Escrita

Fonte: Autor.

A UART foi projetada com dois registradores, um de dados, e outro de controle, sendo o último verificado por meio da técnica de polling, causando de perdas na performance do hardware. Os campos do registrador de controle a serem verificados são os [0] e [2], que correspondem a transmissão pronta, e recepção pronta, respectivamente. Dos 32 bits do registrador de controle, apenas 8 são utilizados, sendo que no momento apenas os 4 LSb possuem definição, o restante se retorna 0 em operação de leitura, sendo a escrita

ignorada. Os bits [1] e [3] são controladores pelo hardware, estando apenas a leitura disponível sobre eles. O registrador de dados possui um tamanho de 8 bits, ignorando-se os bits superiores do dado de entrada em operações de escrita. Na leitura, se estende com 0 o dado. A Figura 2.8 contém o registrador de 8 bits que controla a UART, ocultando-se o restante até o bit 31, que retornam 0.

Figura 2.8 – Registrador de controle da UART, R_ND e TX_R indicam recepção e transmissão pronta, respectivamente.

	7	6	5	4	3	2	1	0
	RES				RX_R	R_ND	TX_W	TX_R
Type	RO	RO	RO	RO	RO	R/W	RO	R/W
Reset	0	0	0	0	0	0	0	0

Fonte: Autor.

2.3 SISTEMA DE MEMÓRIA

A memória sendo um dos componentes fundamentais para o funcionamento de um computador, também é um dos maiores causadores de perda de desempenho do mesmo. Sua baixa velocidade de operação é distante da que o núcleo atinge, fazendo com que constantemente o restante do sistema fique em estado ocioso a espera de seus dados. Idealmente, se desejaria uma memória grande o suficiente para armazenar todos os dados do usuário, e com velocidade para acompanhar o núcleo. Na prática, se utiliza de grandes volumes de dispositivos de memória não voláteis como armazenamento permanente dos dados, como o HDD (Hard Drive Disk). Seu custo de armazenamento é vantajoso para uso como memória final, mas durante a operação do sistema, se torna muito lento, sendo necessária uma memória intermediária para a qual o programa em execução deve ser carregado.

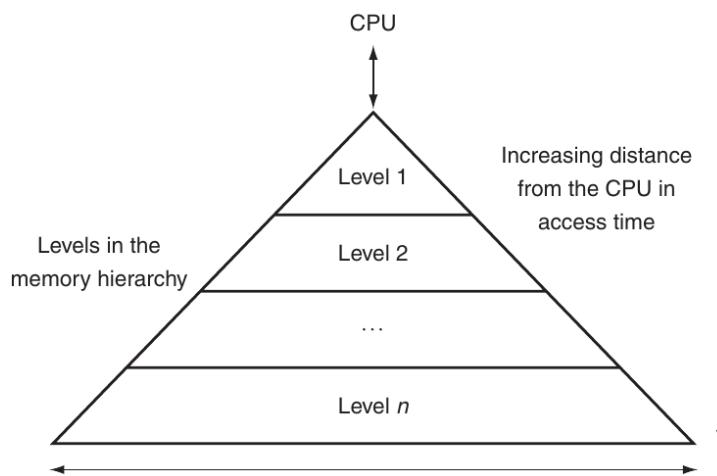
Essa memória principal utilizada na execução de programas é volátil, podendo ser implementada com tecnologias diversas. Comumente ela é referida como DRAM, projetada a base de capacitores, e possuindo grande benefício de área. Mas ainda sim, o processador fica ocioso a espera de dados por ela, que apesar de possuir um equilíbrio bom de área e velocidade, não alcança o núcleo. Se tornando necessária então a implementação de uma memória local nomeada de Cache, que opera o mais próximo possível do processador, e sendo implementada atualmente até dentro do mesmo CI. Sua alta velocidade de operação se reflete no seu custo elevado, e por consequência sua densidade de armazenamento é escassa em comparação com a memória DRAM.

Devido a baixa capacidade de armazenamento da Cache, seu projeto envolve diversas técnicas e princípios a fim de se manter apenas os dados essências a execução do sistema. Os princípios de localidade justificam o uso da Cache, podendo ser descritos como:

- **Localidade Temporal:** se um item for referenciado, a tendência é que seja novamente em breve;
- **Localidade Espacial:** se um item for referenciado, a tendência é que em breve itens próximos a ele sejam referenciados também.

Apesar desses conceitos serem geralmente mais visualizados no estudo da Cache (pelo seu tamanho reduzido), se aplicam a todo o sistema de memória. A localidade espacial é facilmente compreendida ao relembrar como funciona o endereçamento para busca de instruções na memória, dentro do núcleo, que incrementa o PC de forma progressiva ao endereço seguinte. Já a localidade temporal é mais perceptível ao analisar uma sequência de código que realiza operações matemáticas como somatórios (PATTERSON; HENNESSY, 2004). A construção de níveis da memória é dita como Hierarquia da Memória, e pode ser facilmente compreendida pela Figura 2.9, em que o menor armazenamento (e mais rápido) está mais próximo do processador, enquanto o mais lento, mas com maior capacidade, está mais distante.

Figura 2.9 – Hierarquia do sistema de memória.

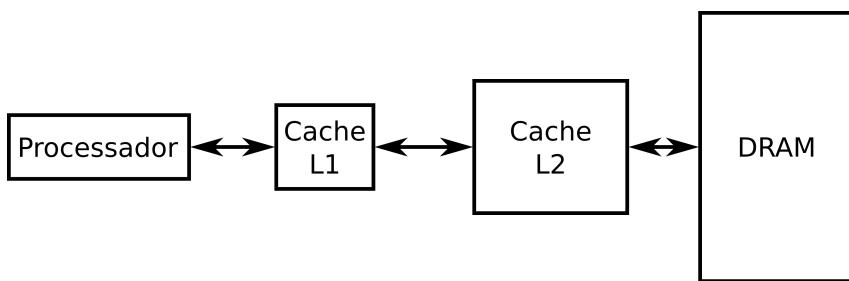


Fonte: Patterson e Hennessy (2004, p. 473).

2.3.1 Cache

A Cache é constituída por tecnologia SRAM, operando extremamente rápido em comparação com outros elementos de memória (externos ao processador), porém custosa em termos de área. Seu hardware pode ser encontrado dentro do mesmo encapsulamento que o processador, a fim de diminuir ainda mais a latência. Seu tamanho é encontrado na ordem de kiBytes, possuindo implementações diversas desde 4 kiB até 64 kiB. Por conta de seu tamanho reduzido, é comum em sistemas grandes a implementação de outros níveis de cache entre o processador e a memória DRAM principal, sendo a cada nível mais próximo da DRAM reduzida sua velocidade, porém aumentada a densidade de armazenamento. O primeiro nível em relação ao núcleo é referenciado como L1, operando próximo a frequência do mesmo, e as demais em ordem crescente L2 e L3. A Figura 2.10 contém a implementação de dois níveis de cache entre o processador e a memória DRAM.

Figura 2.10 – Implementação de dois níveis da memória cache.



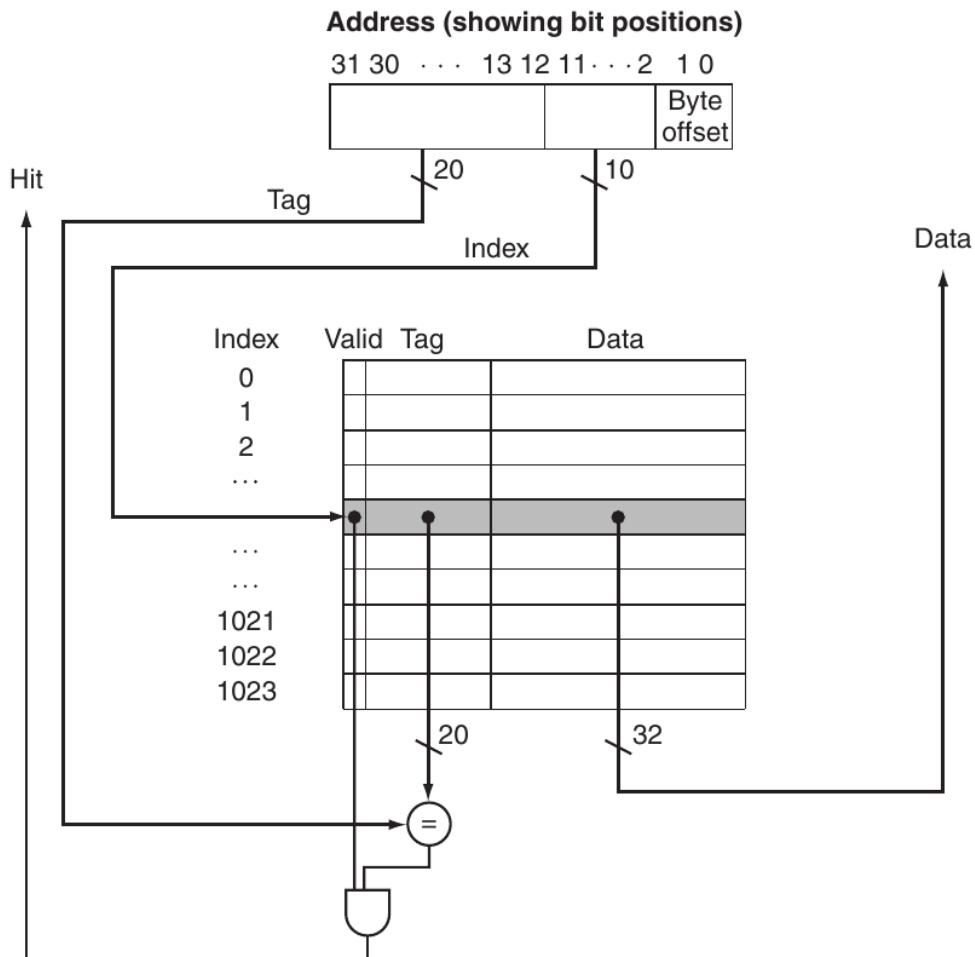
Fonte: Autor.

2.3.1.1 Mapeamento

Como a memória cache possuiu tamanho reduzido em comparação com a final, é necessário a implementação de uma técnica de mapeamento na mesma. Para o processador a cache não existe, e quando se acessa um dado, o objetivo é busca-lo na memória principal. Suponhamos então que se tem 32 bits de endereço, possibilitando 4 GiB a serem acessados, porém a cache é de apenas 1 kiB, utilizando apenas 10 bits do endereço. Inevitavelmente, apenas 10 bits serão utilizados como index da cache, e o restante se interpreta como informação para determinar se o dado do endereço lido corresponde ou não ao desejado. O diagrama de blocos da Figura 2.11 demonstra um mapeamento direto, aonde se utiliza os bits [11:2] para acessar a linha da cache, e os bits [31:12] como tag. Essa tag, quando igual (verificada no comparador de saída) indica que o dado ali presente corresponde ao desejado, habilitando então seu uso.

Alguns detalhes devem ser ressaltados a respeito desse exemplo. Primeiro, os 2 LSb do endereço são ignorados, pois a cache armazena palavra (aqui de 32 bits), dife-

Figura 2.11 – Mapeamento direto para uma cache de 1 kiB.



Fonte: Patterson e Hennessy (2004, p. 478).

rente da memória a byte. A informação dos 2 LSb podem ser então implementada como mecanismo de acesso a byte ou meia palavra na cache. O segundo ponto a abordar é o bit de validade armazenado em conjunto com a tag e o dado. Quando em nível 1, indica que o dado na linha acessada é válido, e então considerada-se a tag para comparação. Do contrário, o resultado é que não se encontrou o dado desejado na linha, gerando uma flag hit. Essa flag em nível 0 faz a solicitação ao controle da cache para buscar esse dado faltando, acessando a memória principal. Deixando assim o sistema (que depende dele) em estado ocioso até sua chegada. O campo de validade também é utilizado para se invalidar a cache intencionalmente em casos específicos, como por exemplo, na inicialização do sistema, em que seu estado é geralmente desconhecido, colocando todos os bits em nível 0.

Caso o bit de validade esteja em nível 1, porém a tag não seja equivalente, também se gera uma solicitação de busca para o controle, sendo necessário descartar o dado presente na linha a fim de se alocar espaço ao novo que irá chegar. Como é perceptível, esse mapeamento não tem nenhum controle sobre aonde armazenar dados novos que

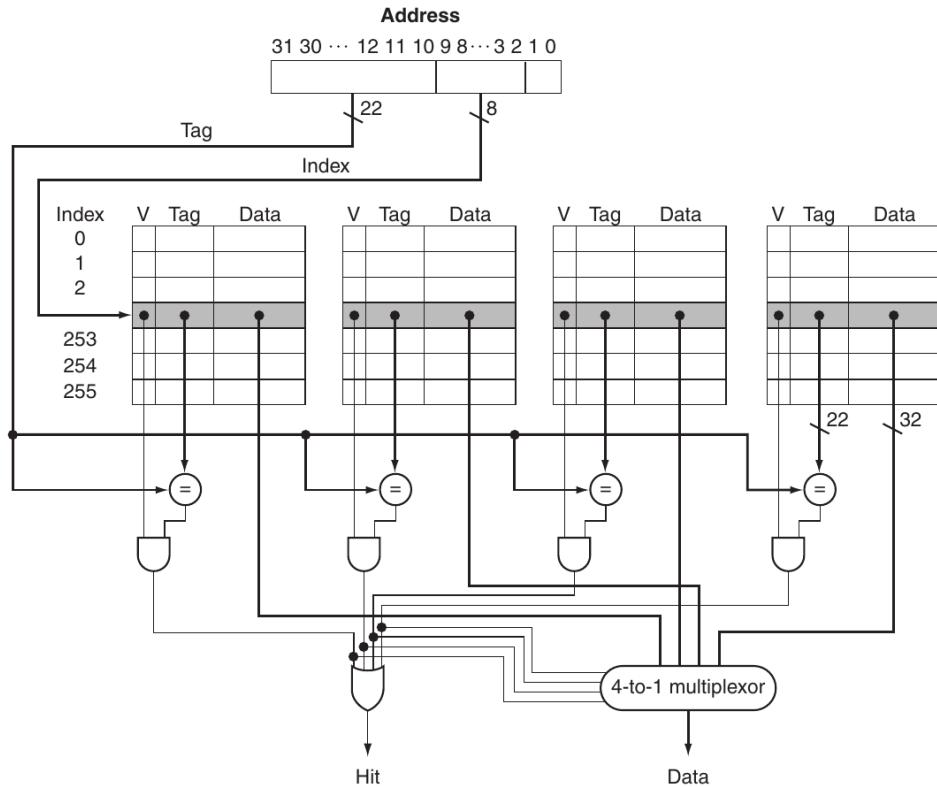
são buscados da memória principal, podendo causar diversos miss ($hit = 0$) de acesso indesejados. Porém, seu hardware de controle é relativamente simples e pequeno, contendo apenas um comparador para a linha selecionada, e sem necessidade de algoritmos de substituição. Diferentes implementações podem armazenar mais de uma palavra por linha, nesse caso, supondo 2 palavras de 32 bits e um mesmo tamanho total de cache (1 kiB), o index iria se reduzir em 1 bit, removendo-se seu LSb (bit 3), que agora é utilizado para selecionar quais das duas palavras da linha se deseja.

Outro tipo de mapeamento, que se encaixa como o oposto do direto, seria o totalmente associativo. Nessa implementação se tem a possibilidade de alocar-se um dado em qualquer uma das linhas da cache. Como consequência, ao acessar a cache é necessária a verificação de todo o seu armazenamento (pois o dado pode estar em qualquer local), que se realizado de forma serial é muito lenta. E ao realizar-se de forma paralela, acrescenta muito em hardware ao utilizar um comparador para cada tag. Ainda se tem a necessidade de um algoritmo de substituição para determinar aonde alocar dados novos buscados da memória. Algoritmos geralmente utilizados são FIFO (first-in first-out), LIFO (last-in first-out), LRU (least recently used) e LFU (least frequently user).

Um meio termo entre o mapeamento direto e o totalmente associativo, que busca não acrescentar tanto em hardware no seu acesso mas ainda sim não limitar o posicionamento de dados a um único local, é o mapeamento associativo por conjunto. Na associação por conjunto, é possível de se alocar um dado em pelo menos 2 posições diferentes, no caso de se limitar a 1 posição, o mapeamento é correspondente a um direto. Cada linha é referida como um conjunto aonde se armazenam blocos, que constituem uma tag e seu(s) dado(s), logo, o mapeamento associativo por conjunto possui pelo menos dois blocos por conjunto, excluindo o equivalente ao mapeamento direto. A substituição de dados é determinada entre um desses blocos da linha acessada. O número de blocos associados a um conjunto é expresso pelos termos one-way (1 conjunto = mapeamento direto), two-way (dois conjuntos), four-way (quatro conjuntos), por exemplo. O mapeamento da Figura 2.12 demonstra uma associação por conjunto four-way, para uma memória de 1 kiB. É visível o acréscimo em comparadores e MUX, porém ainda se tem um ganho relevante em relação a implementação totalmente associativa, enquanto se mantém um grau de liberdade sobre aonde posicionar novos dados. Um ponto crítico em relação ao acréscimos de associação se faz na tag, pois a redução de linhas na cache tem como consequência o aumento da tag armazenada na memória.

Uma análise é realizada por Kim e Song (2017), aonde se verifica o desempenho de uma memória cache variando-se seu tamanho de 2 kiB, até 16 kiB. A cache é separada em duas, uma para armazenamento de instruções, e outra para dados. Os testes foram avaliados sobre a FPGA da Xilinx ZYNQ-7000, com rotinas que consistiam em acessos de leitura/escrita aleatórios na memória, e também sequenciais. Ao aumentar a cache de 2 kiB para 16 kiB, constatou-se um aumento de 273.8% na performance em escritas

Figura 2.12 – Mapeamento associativo por conjunto 4-way, 1 kiB.



Fonte: Patterson e Hennessy (2004, p. 503)

aleatórias, e 214.4% para leituras aleatórias. Para acessos sequênciais, verificou-se um aumento de 281.6% e 313.3%, em escrita e leitura, respectivamente.

Archana e Kashwan (2016) fazem uma verificação do desempenho em termos de consumo de potência para dois mapeamentos diferentes, uma cache com mapeamento direto, e outra associativa por conjunto. Justificou-se o mapeamento direto como uma alternativa vantajosa em termos de consumo de potência, devido a sua baixa complexidade de implementação. Pois sua indexação e verificação direta da tag requer um controle simplificado, sem uso de algoritmos de substituição. As memórias utilizadas eram acessadas por endereços de 8 bits, sendo o modelo associativo por conjunto um 4-way.

2.3.2 Coerência de Dados

O acesso a cache para leitura não altera os dados armazenados na memória, sendo assim, nenhuma modificação é propagada para a memória principal, porém para escrita o caso é diferente. Ao se armazenar um dado na cache por meio de uma operação do núcleo, temos que uma mesma posição de memória alocada na cache possuí um valor diferente do seu endereço equivalente na memória principal. Essa diferença é referenciada

como incoerência, aonde se tem a necessidade de manter os dados com o mesmo valor. Duas formas básicas podem ser citadas para tratar de operações de escritas na cache, write-through e write-back. A técnica de write-through é simples, consiste em escrever um dado na cache, e simultaneamente já começar a escrita desse mesmo dado na memória principal, para manter a coerência. Evidentemente, esse processo é lento, considerando que pelo princípio de localidade temporal esse mesmo dado será acessado novamente, utilizando o caminho lento da memória principal diversas vezes. Ainda sim, essa técnica mantém com sucesso a coerência entre as memórias.

Na implementação de write-back se evita a escrita constante na memória, pois como dito, o princípio de localidade sugere que o mesmo será utilizado novamente. Dessa forma, o dado somente será escrito de volta para a memória principal quando ele for removido da cache por um miss na sua posição. Essa abordagem elimina diversos acessos lentos a memória principal e também poupa energia utilizada no armazenamento constante da memória. Para uso do write-back é necessário armazenar uma informação extra na cache em conjunto com o bit de validade de cada dado. Essa nova flag em nível 1 indica a modificação do dado, interpretando-se então que o endereço correspondente da DRAM está desatualizado e deve ser escrito. Para o nível 0 não se tem a necessidade de escrever de volta na memória, evitando-se assim escritas desnecessárias que iriam ocupar o barramento de troca de dados entre a cache e a memória principal.

A coerência deve ser mantida entre cada nível de memória implementada, e não somente entre a memória principal, utilizando-se essas lógicas entre caches de nível L1 e L2, L2 e L3, e assim por diante. A necessidade de coerência aqui abordada é simples, mas em sistemas mais complexos diversos outros fatores podem causar a incoerência de dados, como múltiplos núcleos (cada um com sua memória cache) acessando o mesmo dado. Também existe a possibilidade de um DMA (Direct Memory Access) alterar um dado da memória.

2.4 PLACA NEXYS3

A placa Nexys3 projetada pela Digilent Adept conta com a FPGA da família Spartan-6 da Xilinx, dispositivo XC6SLX16, com alta capacidade lógica disponível no uso de projetos digitais. Além da FPGA, se tem disponível também um cristal oscilador de 100 MHz para uso como sinal de relógio, diversos componentes de comunicação como portas USB, Ethernet, UART (Universal Asynchronous Receiver/Transmitter), entre outros. Dentre esses componentes, o destaque por ser utilizado no projeto se da a FPGA, a UART para comunicação com o hardware interno do processador alvo, e a memória de 16 MByte externa da placa (DIGILENT, 2013).

2.4.1 FPGA Spartan-6

A FPGA da Xilinx Spartan-6 XC6SLX16 possui além de sua lógica reconfigurável por meio de LUTs (Look-Up-Tables), diversos componentes dedicados como BRAMs (Block RAM), DSP (Digital Signal Processor), IOBUF (Buffers I/O), e diversos outros. Os buffers I/O são dedicados para os pinos do CI, sendo internamente utilizado multiplexados para implementar buffers tri-state descritos em HDL. O número de slices, cada um contendo 6-input LUT com 8 FF (flip-flop), são de 2.278, totalizando assim 18.224 FFs.

2.4.1.1 Memória

O uso de memória na FPGA pode ser implementada de três formas: BRAMs, RAM distribuída, ou então puramente por LUTs. O uso de RAM distribuída na realidade, utiliza LUTs, porém é uma configuração específica das mesmas, em que uma única LUT representa uma memória 16x1 bit. Sua configuração utiliza uma escrita síncrona, e leitura síncrona ou assíncrona, sendo que a memória não possui reset. O uso de LUTs para implementar uma memória que não se encaixa como RAM distribuída é consequência justamente de características como o reset, ou escrita assíncrona. O uso de RAM distribuída tem ganho de performance de área e velocidade em relação a puramente por LUTs.

Os blocos de RAM são componentes internos da FPGA dedicados a implementar grandes armazenamentos de memória, sendo disponibilizado um total de 32 BRAMs na Spartan-6, cada uma com até 2 KByte, totalizando uma capacidade de 64 kBytes. Diferente da RAM distribuída, as BRAMs são mais restritas no seu funcionamento, sendo totalmente síncrona (leitura e escrita), e com um número de portas variando de uma a duas. As BRAMs estão posicionadas em locais fixos internamente no CI, afetando a lógica de síntese e implementação dos hardwares descritos em HDL, sendo ideal seu uso então quando se necessita de bastante armazenamento. Em casos em que a memória é pequena, pode ser mais viável o uso de RAM distribuída para possibilitar a ferramenta de síntese mais liberdade de otimização (XILINX, 2011; XILINX, 2009).

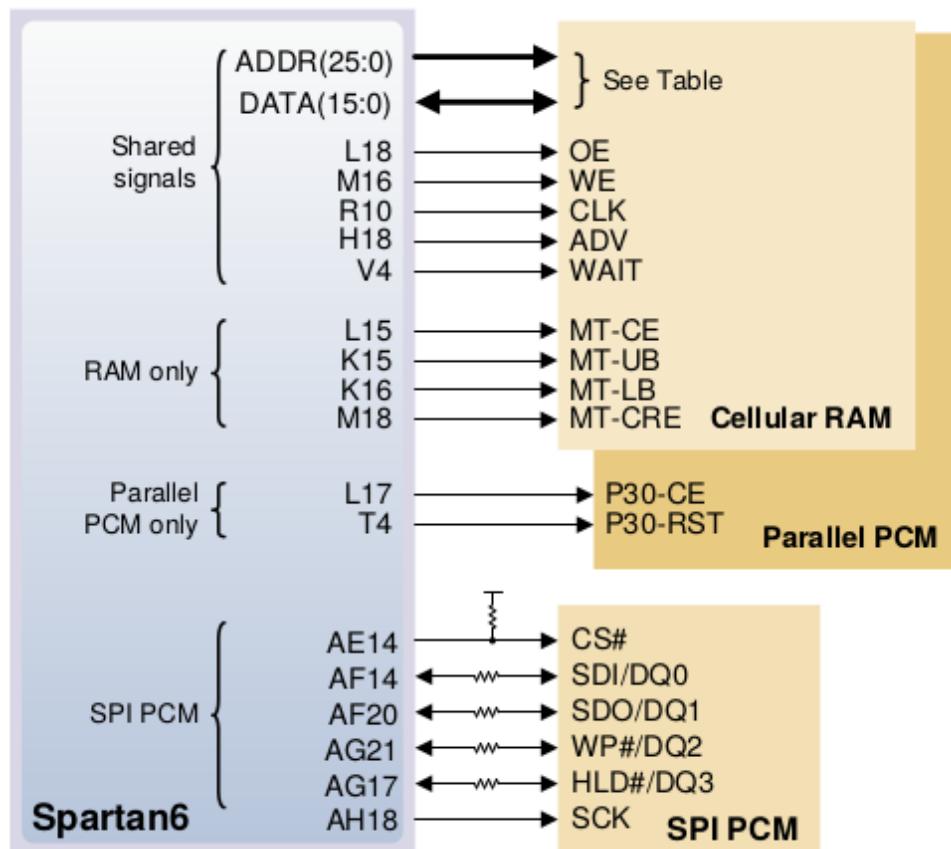
Assim como a RAM distribuída, BRAMs não possuem reset. Ambos os tipos de memória podem ser descritos em HDL com um valor inicial que é carregado para a memória durante a configuração da FPGA, durante a bitstream.

2.4.2 Cellular RAM

A placa Nexys3 conta com três memórias, sendo duas voláteis, e uma não volátil, todas de 16 MBytes. A memória RAM presente da Micron, de número M45W8NW16,

possui endereçamento a palavra de 16 bits, com acessos de 70 ns no modo assíncrono, e seu modo page (variante do assíncrono). Também conta com operações de burst no modo síncrono, podendo funcionar em até 80 MHz. Durante a operação assíncrona a memória possui um controle simplificado de refresh (MICRON,). Por padrão, durante a inicialização do dispositivo de memória, o mesmo é configurado no modo assíncrono sem operações page. A Figura 2.13 demonstra as conexões da FPGA Spartan-6 com a Cellular RAM na placa Nexys3, e também das memórias PCM (voláteis).

Figura 2.13 – Conexões da FPGA Spartan-6 com a Cellular RAM na placa Nexys3.



Fonte: Digilent (2013, p. 10).

O sinal MT-CRE (control register enable) representa a habilitação de acesso aos registradores de controle da RAM, que definem seu modo de operação, entre outras características possíveis, como o refresh parcial de seu array, com opções como 1/4 ou 1/2. Funcionalidade útil para poupar energia em casos de uso limitado da memória. Seu nível ativo é 1, nesse intervalo se realiza um acesso assíncrono normal a memória (pois o CI é inicializado assim), no qual a porta de dados é ignorada. Os dados a serem escritos estão todos contidos no campo de endereço, possuindo bits que especificam o registrador a ser acessado entre: BCR (Bus Configuration Register), RCR (Refresh configuration Register) e DIDR (Device Identification Register).

2.4.2.1 Registradores de Controle

O registrador BCR é responsável pela configuração do modo de operação do dispositivo, entre assíncrono e síncrono. Dentro do modo síncrono ainda é possível configurar diversas outras funcionalidades durante uma operações burst, como o comprimento da mesma, ou até mesmo um acesso contínuo, por padrão ele se inicializa no modo assíncrono. No registrador BCR se encontra as opções de refresh do CI, como refresh parcial de seu array, a fim de poupar energia, ou até a habilitação do modo DPD (Deep Power-Down) no qual o CI desabilita o seu refresh, a fim de entrar em estado de baixo consumo. Também é nesse registrador que se encontra a habilitação do modo assíncrono page. No registrador DIDR se encontram apenas informações do dispositivo, como sua geração, densidade (tamanho em MBits), entre outras utilidades.

2.4.2.2 Modo Assíncrono

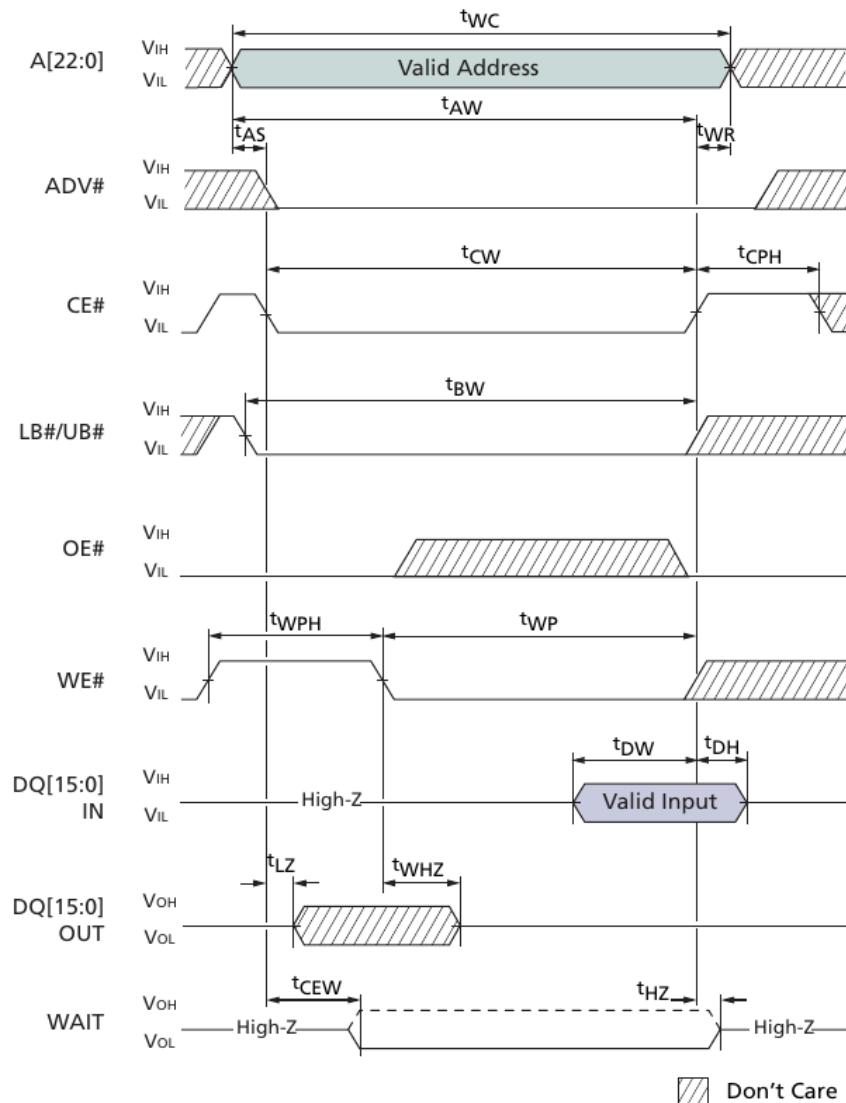
No modo assíncrono as portas CLK (clock) e WAIT não acrescentam nenhuma informação, sendo CLK fixada em nível '1' ou 0, e o sinal WAIT apenas ignorado. A porta ADV (address valid) pode ser mantida em nível 0 durante todo o acesso para leitura/escrita da memória. Seu acesso para leitura ou escrita é iniciado quando a porta MT-CE (chip enable) é colocada em nível 0, após isso, ao deixar em nível ativo OE (output enable), também em nível 0, se faz um acesso de leitura. A entrada WE (write enable) caracteriza a escrita, ativo em '0', quando esse esta ativo é possível ignorar o estado de OE, pois ele possui prioridade. Com armazenamento de 2 bytes por endereço, se tem os sinais de controle MT-UB e MT-LB (upper byte e lower byte) que possibilitam o acesso a bytes na memória, sendo ativos em nível 0.

Durante operações de escrita, o dado a ser armazenado é registrado na memória durante a primeira borda de subida que ocorrer entre os sinais MT-CE, WE, MT-UP ou MT-LB, conforme o diagrama de tempo apresentado na Figura 2.14. O sinal MT-CE não pode permanecer ativo por mais 4 us devido a questões de refresh da memória, correndo o risco de perda de dados caso o tempo não seja respeitado.

2.4.2.3 Modo Page

O modo page funciona da mesma forma que o assíncrono durante operações de escrita, a diferença está no acesso para leitura. A operação aproveita-se do princípio de que endereços próximos ao atual podem ser acessados com uma latência reduzida, sendo organizada então a memória em páginas de 16 palavras. Após o primeiro acesso a página,

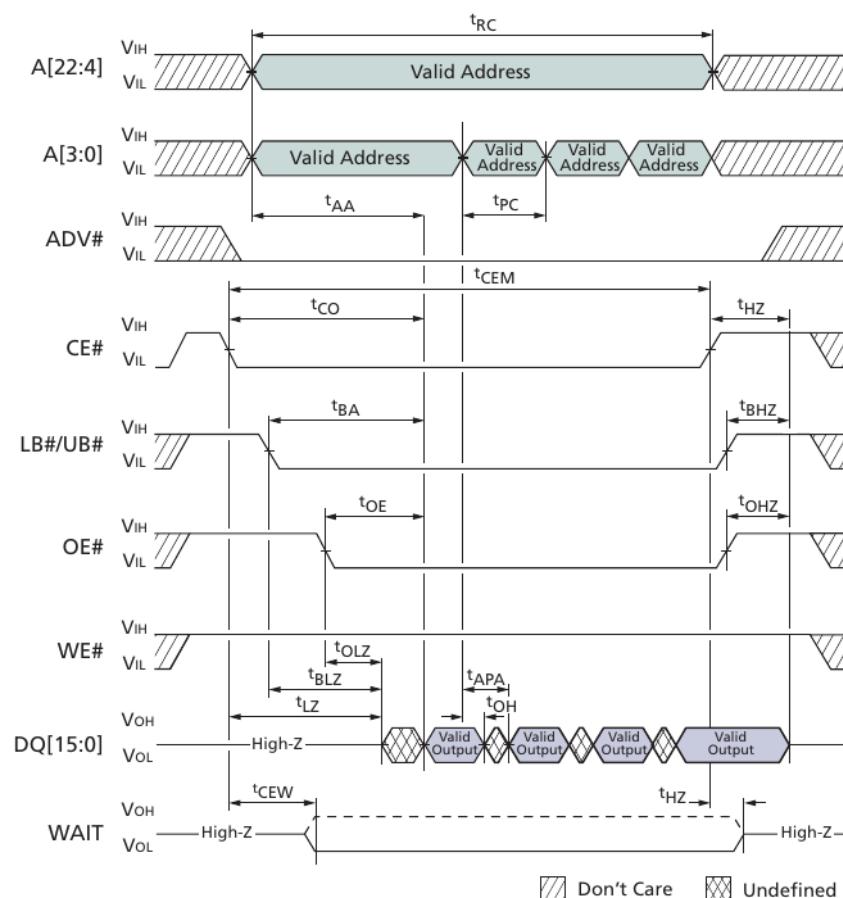
Figura 2.14 – Diagrama de tempo da Cellular RAM no modo assíncrono durante uma operação de escrita, controlada por MT-CE, designado como CE# no diagrama.



Fonte: Micron (, p. 51).

é possível então realizar a leitura dos endereços próximos em questão de 20 ns ao alterar apenas os 4 LSB do endereço. Qualquer alteração nos bits superiores irá finalizar o acesso a página. A Figura 2.15 demonstra um diagrama de tempo, aonde o primeiro acesso assíncrono é o padrão, e os demais aproveitam do modo page para realizar acessos rápidos a endereços próximos.

Figura 2.15 – Diagrama de tempo da Cellular RAM no modo assíncrono page.



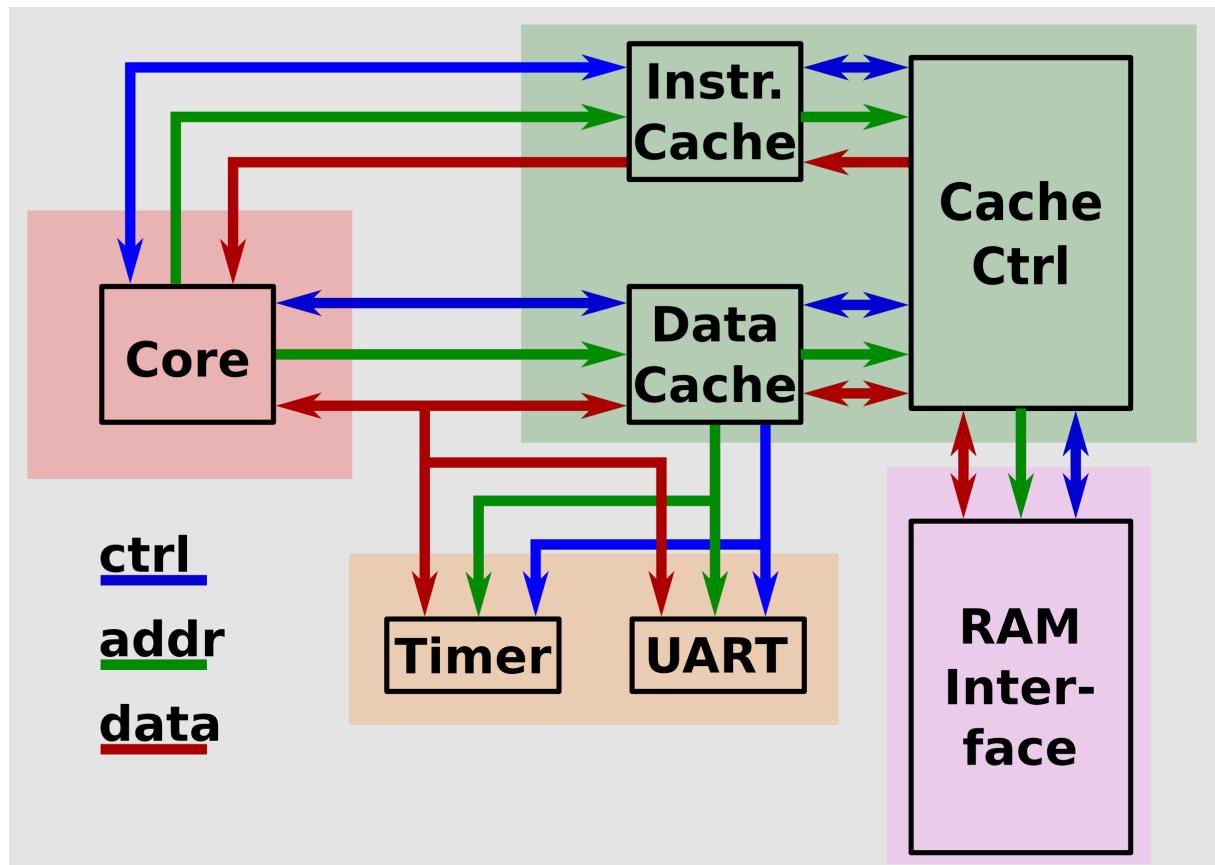
Fonte: Micron (, p. 44).

3 METODOLOGIA E ESPECIFICAÇÕES

Neste capítulo se apresenta o desenvolvimento do projeto, com esquemáticos, tabelas, e as tomadas de decisões realizadas ao longo do mesmo. Na Seção 3.1 é apresentada a parte de software do projeto, com as instruções implementada da ISA, e o mapeamento de memória utilizado pelo script de link, por exemplo. Na Seção 3.2 demonstra-se as modificações realizadas no núcleo. Na Seção 3.3 se tem o projeto da memória cache do processador, substituindo o esquemático inicial utilizado anteriormente. Na Seção 3.4 é apresentada a interface projetada para controlar a memória RAM disponível na placa nexys 3. Na Seção 3.5 é discutido as mudanças realizadas no controle dos dispositivos I/O internos do processador, a UART e o Temporizador.

A divisão de seções foi realizada conforme os componentes projetados do processador, incluíndo a interface com a memória RAM externa, uma visão geral do hardware é apresentada na Figura 3.1.

Figura 3.1 – Visão geral do processador.



Fonte: Autor.

3.1 PROGRAMAÇÃO

Essa seção descreve a parte de software do projeto, contendo todas as instruções implementadas da ISA do RISC-V (WATERMAN; ASANOVIć, 2017a; WATERMAN; ASANOVIć, 2017b). Também apresenta o processo de compilação para programas escritos em C/ASM, e informações a respeito do mapeamento de memória projetado para se utilizar com o processador, incluindo características do script de link.

3.1.1 Instruções

Implementou-se todas as instruções da base RV32I do RISC-V com exceção das utilizadas para realizar chamadas a níveis superior de privilégio, ou então sistemas supervisores, EBREAK e ECALL. Também ignorou-se as instruções FENCE, que são utilizadas para manter a coerência entre diferentes módulos de memória, um exemplo seria múltiplos núcleos, cada um com sua própria cache L1 não compartilhada. Mais detalhes sobre cada instrução podem ser encontrados no manual a nível de usuário e privilegiado (WATERMAN; ASANOVIć, 2017a; WATERMAN; ASANOVIć, 2017b).

3.1.1.1 Operação Register-Register

Instruções de operações entre dois registradores do banco, possuem o formato da Figura 3.2.

Figura 3.2 – Formato de instruções Register-Register.

31	25 24	20 19	15 14	12 11	7 6	0
funct7	rs2	rs1	funct3	rd	opcode	

7 5 5 3 5 7

Fonte: Adaptado de Waterman e Asanović (2017a, p. 15).

- **ADD**: realiza a soma entre os regs. rs1 e rs2, armazenando o resultado em rd;
- **SUB**: realiza a subtração de rs2 em rs1, armazenando o resultado em rd;
- **SLL**: realiza o deslocamento lógica a esquerda de rs1 pelo valor contido no regs. rs2, armazenando o resultado em rd;
- **SRL**: realiza o deslocamento lógica a direita de rs1 pelo valor contido no regs. rs2, armazenando o resultado em rd;
- **SRA**: realiza o deslocamento aritmético a direita de rs1 pelo valor contido no regs. rs2, armazenando o resultado em rd;

- **SLT**: armazena o valor decimal 1 em rd caso rs1 seja menor que rs2, considerando com sinal;
- **SLTU**: armazena o valor decimal 1 em rd caso rs1 seja menor que rs2, considerando sem sinal;
- **OR**: realiza a operação lógica OR bit a bit entre os regs. rs1 e rs2, armazenando o resultado em rd;
- **AND**: realiza a operação lógica AND bit a bit entre os regs. rs1 e rs2, armazenando o resultado em rd;
- **XOR**: realiza a operação lógica XOR bit a bit entre os regs. rs1 e rs2, armazenando o resultado em rd.

3.1.1.2 Operação Register-Immediate

Instruções de operações entre um registrador do banco e o campo imediato instrução, possuem o formato da Figura 3.3. Todos os sinais imediatos são estendidos para 32 bits pelo MSB, que em todo o tipo de instrução está posicionado no bit [31].

Figura 3.3 – Formato de instruções Register-Immediate.

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12	5	3	5	7	

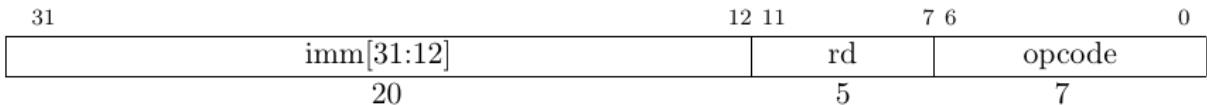
Fonte: Adaptado de Waterman e Asanović (2017a, p. 13).

- **ADDI**: realiza a soma entre o regs. rs1 e o imediato, armazenando o resultado em rd;
- **SLLI**: realiza o deslocamento lógica a esquerda de rs1 pelo valor contido no imediato, armazenando o resultado em rd;
- **SRLI**: realiza o deslocamento lógica a direita de rs1 pelo valor contido no imediato, armazenando o resultado em rd;
- **SRAI**: realiza o deslocamento aritmético a direita de rs1 pelo valor contido no imediato, armazenando o resultado em rd;
- **SLTI**: armazena o valor decimal 1 em rd caso rs1 seja menor que o imediato, considerando com sinal;
- **SLTIU**: armazena o valor decimal 1 em rd caso rs1 seja menor que rs2, considerando sem sinal;
- **ORI**: realiza a operação lógica OR bit a bit entre o regs. rs1 e o imediato, armazenando o resultado em rd;

- **ANDI**: realiza a operação lógica AND bit a bit entre o regs. rs1 e o imediato, armazenando o resultado em rd;
- **XORI**: realiza a operação lógica XOR bit a bit entre o regs. rs1 e o imediato, armazenando o resultado em rd.

As instruções LUI e AUIPC possuem um formato próprio, demonstrado na Figura 3.4.

Figura 3.4 – Formato das instruções LUI e AUIPC.



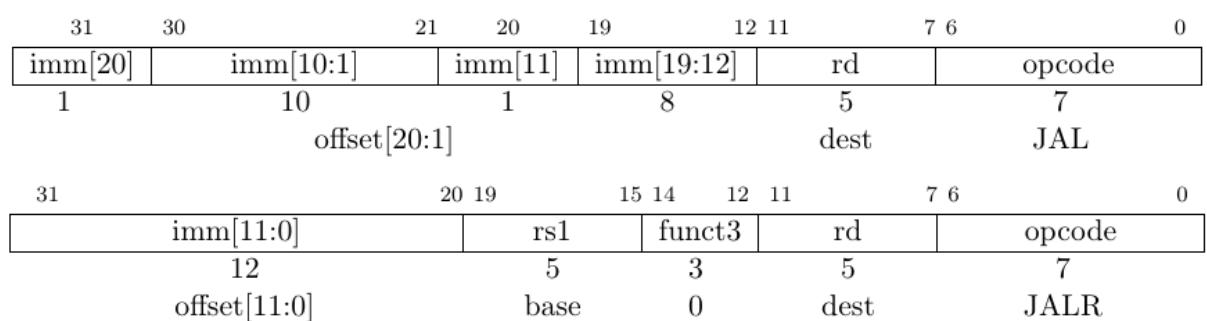
Fonte: Adaptado de Waterman e Asanović (2017a, p. 14).

- **LUI**: armazena o campo imediato de 20 bits na parte superior do regs. rd, deixando em 0 os 12 bits inferiores;
- **AUIPC**: soma campo imediato de 20 bits a parte superior do PC (atual da instrução), armazenando o resultado em rd.

3.1.1.3 Desvios Incondicionais

As instruções de desvio incondicionais se resumem a JAL e JALR, com seus formatos demonstrados na Figura 3.5.

Figura 3.5 – Formato de instruções de desvio incondicional.



Fonte: Adaptado de Waterman e Asanović (2017a, p. 16).

- **JALR**: soma o campo imediato ao valor de PC (atual da instr.), realizando um desvio incondicional ao endereço obtido, enquanto armazena o endereço da instrução em sequência ao jump (PC+4) no registrador rd;

- **JALR**: soma o campo imediato ao valor do regis. rs1, realizando um desvio incondicional ao endereço obtido, enquanto armazena o endereço da instrução em sequência ao jump (PC+4) no registrador rd.

3.1.1.4 Desvios Condicionais

As instruções de desvio condicionais possuem o formato demonstrados na Figura 3.6, os endereços alvos de desvio são obtidos ao somar o campo imediato com o valor de PC (da instr.).

Figura 3.6 – Formato de instruções de desvio condicional.

31	30	25 24	20 19	15 14	12 11	8	7	6	0
imm[12]	imm[10:5]	rs2	rs1	funct3	imm[4:1]	imm[11]		opcode	

1 6 5 5 3 4 1 7

Fonte: Adaptado de Waterman e Asanović (2017a, p. 17).

- **BEQ**: realiza o desvio se o regis. rs1 for igual ao regis. rs2;
- **BNE**: realiza o desvio se o regis. rs1 for diferente do regis. rs2;
- **BLT**: realiza o desvio se o regis. rs1 for menor que o regis. rs2, considerando com sinal;
- **BLTU**: realiza o desvio se o regis. rs1 for menor que o regis. rs2, considerando sem sinal;
- **BGE**: realiza o desvio se o regis. rs1 for maior ou igual que o regis. rs2, considerando com sinal;
- **BGEU**: realiza o desvio se o regis. rs1 for maior ou igual que o regis. rs2, considerando sem sinal;

3.1.1.5 Load e Store

As instruções de leitura/escrita na memória possuem o formato da Figura 3.7. Todas as instruções de store e load utilizam o resultado da soma do regis. rs1 com o imediato como endereço de acesso a memória.

- **LW**: armazena no regis. rd o dado de 32 bits contido no endereço de memória acessado;
- **LH**: armazena no regis. rd o dado de 16 bits contido no endereço de memória acessado, extendido pelo MSB para 32 bits;

Figura 3.7 – Formato de instruções Load/Store.

31	20 19	15 14	12 11	7 6	0
imm[11:0]	rs1	funct3	rd	opcode	
12 offset[11:0]	5 base	3 width	5 dest	7 LOAD	

31	25 24	20 19	15 14	12 11	7 6	0
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	
7 offset[11:5]	5 src	5 base	3 width	5 offset[4:0]	7 STORE	

Fonte: Waterman e Asanović (2017a, p. 19).

- **LHU**: armazena no regs. rd o dado de 16 bits contido no endereço de memória acessado, extendido com 0s para 32 bits;
- **LB**: armazena no regs. rd o dado de 8 bits contido no endereço de memória acessado, extendido pelo MSB para 32 bits;
- **LBU**: armazena no regs. rd o dado de 8 bits contido no endereço de memória acessado, extendido com 0s para 32 bits;
- **SW**: escrever na memória o dado completo (32 bits) contido no regs. rs2;
- **SH**: escreve na memória a meia palavra inferior (16 bits) do dado contido no regs. rs2;
- **SB**: escreve na memória o byte inferior (8 bits) do dado contido no regs. rs2.

3.1.1.6 Instruções de Controle

As instruções de acesso a registradores de controle/status do núcleo possuem o formato da Figura 3.8, sendo válido também para a instrução MRET. Todas as operações realizadas por instruções CSR fazem a leitura do valor antigo contido no registrador, armazenando em rd, enquanto se escreve um novo valor contido no registrador rs1. A instrução MRET realiza um desvio incondicional, contendo os campos rs1 e rd no valor 0, logo seu acesso aos registradores CSR é indiferente, já que x0 é a constante 0.

- **CSRRW**: armazena o valor contido em rs1 no CSR, enquanto armazena o valor antigo de CSR em rd;
- **CSRRC**: utiliza o valor contido em rs1 como uma máscara para uma operação OR com o CSR, enquanto armazena o valor antigo de CSR em rd;
- **CSRRS**: utiliza o valor contido em rs1 como uma máscara para uma operação AND com o CSR, enquanto armazena o valor antigo de CSR em rd;

Figura 3.8 – Formato de instruções CSR e MRET.

31	20 19	15 14	12 11	7 6	0
csr	rs1	funct3	rd	opcode	
12	5	3	5	7	

Fonte: Waterman e Asanović (2017a, p. 22).

- **CSRRWI**: armazena o valor do campo rs1 extendido com 0s no CSR, enquanto armazena o valor antigo de CSR em rd;
- **CSRRSI**: utiliza o valor do campo rs1 extendido com 0s como uma máscara para uma operação OR com o CSR, enquanto armazena o valor antigo de CSR em rd;
- **CSRRCI**: utiliza o valor do campo rs1 extendido com 0s como uma máscara para uma operação AND com o CSR, enquanto armazena o valor antigo de CSR em rd;
- **MRET**: desvia o valor do PC para o endereço contido no CSR mepc, enquanto reabilita as interrupções globais no CSR mstatus.

3.1.2 Compilação em C/ASM

Para se obter um programa a ser executado no hardware projetado, se utiliza algumas flags em específico em conjunto com o compilador GCC do RISC-V:

- 1) -march=rv32i: especifica o uso da base RV32I do RISC-V;
- 2) -mabi=ilp32: especifica o uso de convenções de dados do tipo inteiro de 32 bits do RISC-V;
- 3) -mstrict-align: especifica que o hardware/software não permite acessos desalinhados a memória;
- 4) -mpreferred-stack-boundary=3: especifica o uso da pilha no processo de alocação de espaço para dados, no caso, 2 na potência de 3 bytes, totalizando 8 bytes;
- 5) -nostartfiles -T path_to_link/link.ld: especificações para uma aplicação sem SO.

As flags 1 e 2 são utilizadas para definir a arquitetura (instruções) e a ABI do hardware, enquanto a 3 se utiliza para manter alinhado o acesso a memória. A flag 4 é necessária para reduzir o tamanho de dados que a pilha ocupa quando está alocando espaço para novos dados, por padrão o compilador do RISC-V utiliza o maior dado possível, 128 bits para a base RV128I, indiferente da arquitetura definida na flag 1. Com essa flag podemos então reduzir o espaço utilizado para 64 bits (8 bytes), que é o mínimo suportado pelo compilador. As flags 5 são utilizadas para que o programa possa ser utilizado em um sistema bare-metal, sem sistemas operacionais por exemplo, em que se especifica

também o caminho para o script de link. Com essas opções é possível de se obter um programa que execute corretamente no processador, sendo necessário no final extrair o arquivo hexadecimal da intel para envia-lo a FPGA. O comando de compilação completo ficaria:

```
linha 1 => riscv64-unknown-elf-gcc -march=rv32i -mabi=ilp32
-mstrict-align -mpreferred-stack-boundary=3 -nostartfiles
-T path_to_link/link.ld -o main
linha 2 => riscv64-unknown-elf-objcopy -O ihex main main.hex
```

3.1.3 Mapeamento da Memória

O script de link que atribui as seções dos programas escrito em C/ASM foi reescrito de forma a utilizar um total de 16 MBytes, correspondendo a capacidade de armazenamento disponível na memória RAM da placa Nexys3. A Figura 3.9 demonstra o novo mapeamento da memória utilizado, com as seções e seus segmentos internos aplicados no momento de ligação dos códigos. Dentre as alterações realizadas, destaca-se a adição das seções de tabela de vetores para exceções e interrupções, `m_excp_vector` e `m_intr_vector` respectivamente, assim como da seção principal de tratamento de traps, `m_trap_handler`. O termo trap é utilizado para se referir a ocorrência tanto de interrupções, quanto exceções, conforme mencionado por Waterman e Asanović (2017a).

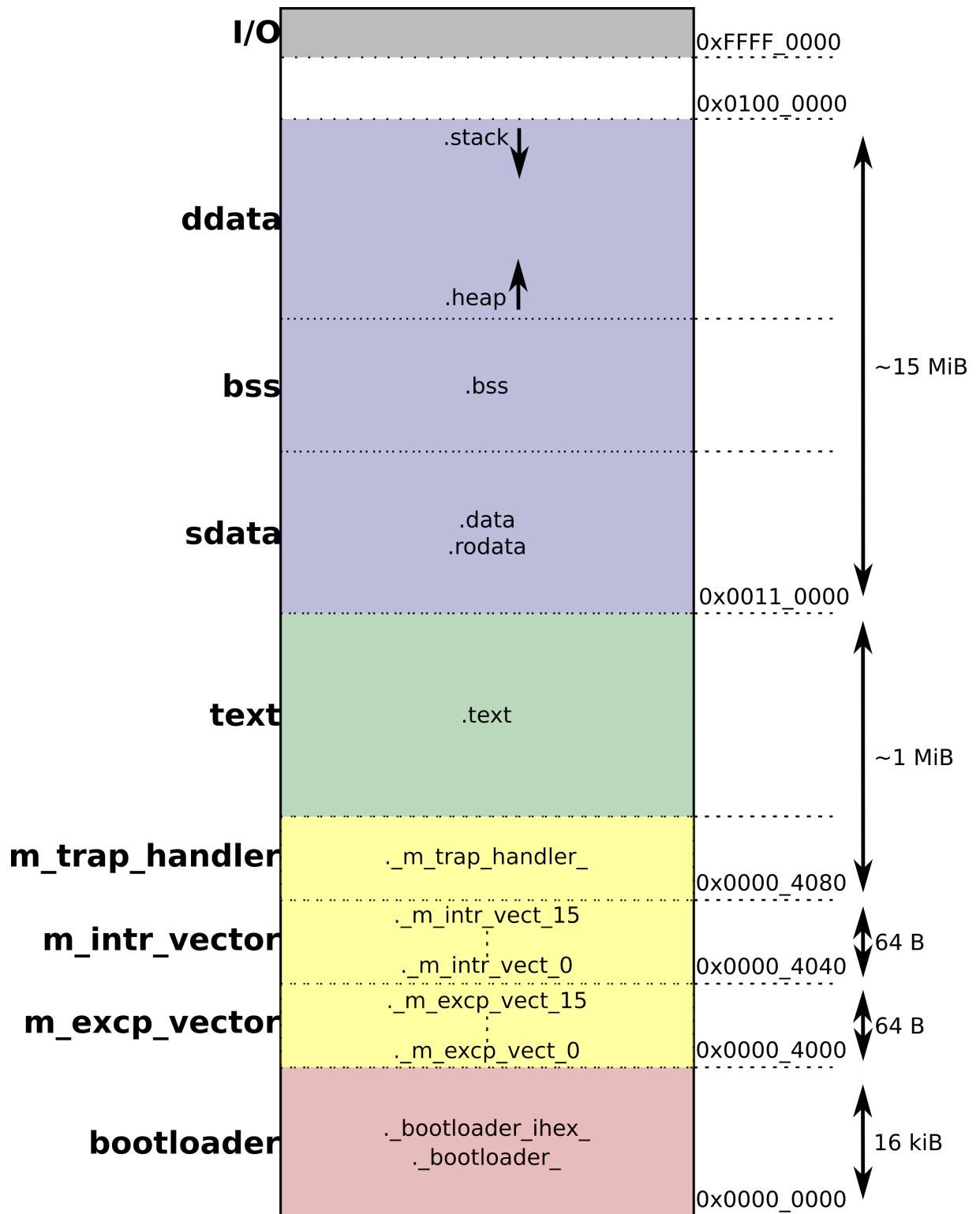
O prefixo "m" nas seções e segmentos relacionados a traps se referem ao nível de privilégio M (Machine), as subseções a seguir descrevem em mais detalhes cada seção da memória, e seus segmentos internos. Uma observação é que cada segmento interno de uma seção corresponde a uma seção de entrada do script de link, ou seja, caso se deseje escrever um programa em ASM dentro da seção bootloader, que comece no segmento `_bootloader_ihex_`, deve-se declarar a mesma como uma seção com atributos de execução e alocação ("ax"). Em ASM e C a declaração ficaria:

```
### ASM Code Example
.section ._bootloader_ihex_,"ax"

code_start:
# code here

-----
/* C Code Example */
```

Figura 3.9 – Mapeamento da memória implementado no script de link.



Fonte: Autor.

```

void boot_ihex() __attribute__((section(". bootloader_ihex")));
void boot_ihex()
{
/* code here */
}

```

3.1.3.1 Seção Bootloader

O segmento boot foi movido para uma seção própria, a bootloader, que é responsável pela inicialização do dispositivo. Seu tamanho foi definido em 16 kiB, o total disponível na cache de instrução, discutido na Seção 3.3. A seção de bootloader tem por responsabilidade inicializar o stack pointer, que é o registrador dedicado x2 na ABI do RISC-V, com o valor do primeiro endereço após o término da seção de dados dinâmicos, 0x0100_0000. Configura também os registradores de controle/status do núcleo para iniciarem com todas as flags de interrupção/exceção desabilitadas, e armazena no CSR mtvec o endereço alvo para o inicio de tratamento de traps, correspondendo ao segmento _m_trap_handler_, endereço 0x0000_4080.

Configura a UART, dispositivo I/O utilizado no projeto que se encontra na sua respectiva seção I/O, a partir do endereço 0xFFFF_0000. Ajusta a taxa de comunicação inicial para 115200 BD, e desabilita todos os tipos de interrupções que a mesma oferece. Essas ações são realizadas dentro do segmento _bootloader_, o próximo segmento é utilizado para carregar um programa do computador pessoal para a FPGA por meio da UART. Como o nome sugere, _bootloader_ihex_ é composto por funções (escritas em C) que fazem a leitura de um arquivo hexadecimal da intel, obtendo os endereços de cada byte do programa, e os escrevendo na memória principal do sistema. A recepção do arquivo hexadecimal é realizada por meio da UART.

Após o término da escrita do programa de usuário na memória RAM, o segmento _bootloader_ihex_ retorna ao segmento principal do bootloader (_bootloader_), aonde se realiza um jump para o endereço inicial do programa do usuário, fornecido pelo símbolo "main". Caso o símbolo não seja definido em nenhum momento no código C/ASM, o script de link o aloca no inicio da seção de texto para que possa realizar o pulo. Apenas uma tarefa é realizada antes do desvio ao código carregado, que consiste em forçar uma operação WB (write-back) da cache de dados. É necessário realizar essa ação pois o programa recebido pela UART é salvo diretamente na cache de dados, e caso não ocorra a substituição de um dado, a memória principal não irá contér o código. Por consequência, quando a cache de instrução for buscar o programa, irá acabar lendo um dado aleatório, pois o desejado ainda está dentro da cache de dados. A execução do write-back consiste em um simples loop que escreve 0 em todas as linhas da cache de dados, em endereços

com tag diferente da seção de texto, forçando a operação write-back. Para evitar a perda de dados se utiliza a região da memória em que se encontra a stack como endereço alvo do valor 0.

3.1.3.2 Seção m_excp/intr_vector

Seções que contém a tabela de vetores para desvios incondicionais (jumps) para as funções de tratamento de exceções e interrupções. Cada tabela possuí um total de 16 entradas, sendo cada entrada formada por apenas uma instrução, um único desvio. Os tipos de exceções e interrupções são enumerados de 0 a 15, conforme a codificação que identifica a causa do mesmo, verificada no registrador de controle/status mcause do núcleo. As duas tabelas em conjunto ocupam um total de 128 bytes, sendo preenchidas com 0 pelo script de link na ocasião de não serem definidas. Optou-se por não definir um vetor de reset, visto que o hardware possuí apenas uma fonte/tipo de reset do sistema, que coloca o PC no endereço 0, executando novamente o programa do bootloader para que se carregue um novo software.

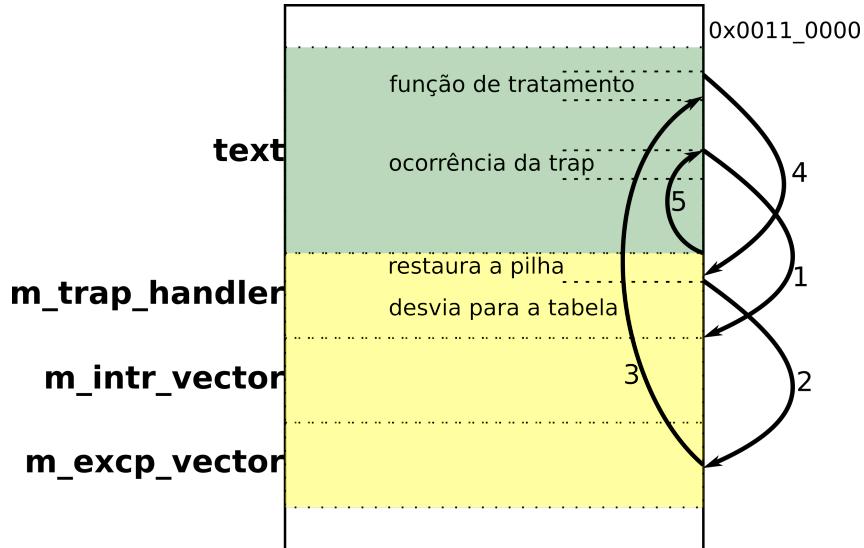
3.1.3.3 Seção m_trap_handler

Essa seção representa a função principal do tratamento de exceção/interrupção, sendo utilizada com endereço inicial de desvio na ocorrência de uma trap. Ao se desviar para o endereço inicial dessa seção, é necessário identificar o que causou a trap, para que se possa realizar um desvio para o segmento da tabela (m_excp_vector ou m_intr_vector) que contém o endereço alvo da função de tratamento. Diferente das seções de tabelas de vetores, essa não possuí tamanho fixo, podendo varia de acordo com a forma que o programador descrever seu funcionamento. De qualquer forma, será um programa relativamente pequeno, que verifica a causa da trap, e então pula para o segmento correspondente de uma das tabelas.

Em uma possível implementação de software para tratamento de trap dentro da seção m_trap_handler, se deseja que a função principal da mesma seja retornada após o tratamento da trap. Pois ela utiliza registradores para determinação da causa de exceção/interrupção, que devem ser restaurados da pilha antes de se voltar ao fluxo normal de execução do programa. A Figura 3.10 demonstra essa forma de uso da função de tratamento principal, na qual o desvio 1 realizado é para o início da seção, aonde se determina a causa, no desvio 2 indo ao segmento (com tamanho de 1 instrução, um jump) da tabela que contém a instrução do desvio 3, redirecionando para a função de tratamento na seção de texto. Após o término do tratamento, se retorna ao próximo endereço do que con-

tinha o desvio na seção `m_trap_handler`, sendo realizada a lógica de restauração da pilha, e por fim fazendo o desvio 5 para a função principal do usuário, finalizando o tratamento.

Figura 3.10 – Fluxo de tratamento de exceções/interrupções.



Fonte: Autor.

3.1.3.4 Seção text e sdata

Na seção de texto está contido todo o programa do usuário, sendo relativamente simples, notando-se apenas que se moveu o segmento rodata (dados constantes), que na implementação anterior encontrava-se em text, para a seção sdata. A seção de dados não inicializados (bss) foi criada separadamente dos dados estáticos (sdata) para que fosse possível inicializar a mesma. Sendo mais claro, quando se energiza a memória principal o estado dos endereços que contém dados não inicializados são indefinidos, porém é comum considerar-se que os mesmos são 0, o que causa um problema, pois o arquivo hexadecimal carregado pela UART não contém informações a respeito de seções vazias. Para evitar isso se escreve sempre um byte 0 na seção por meio do script de link, dessa forma o mesmo não enxerga a seção como vazia, e em complemento do comando FILL a preenche com zeros conforme a necessidade. Assim, a custo de apenas 1 byte de armazenamento da memória (que o link sempre atribui no inicio da seção bss), é possível transmitir por meio do arquivo de extensão ihex as informações a respeito de dados não inicializados.

3.2 NÚCLEO

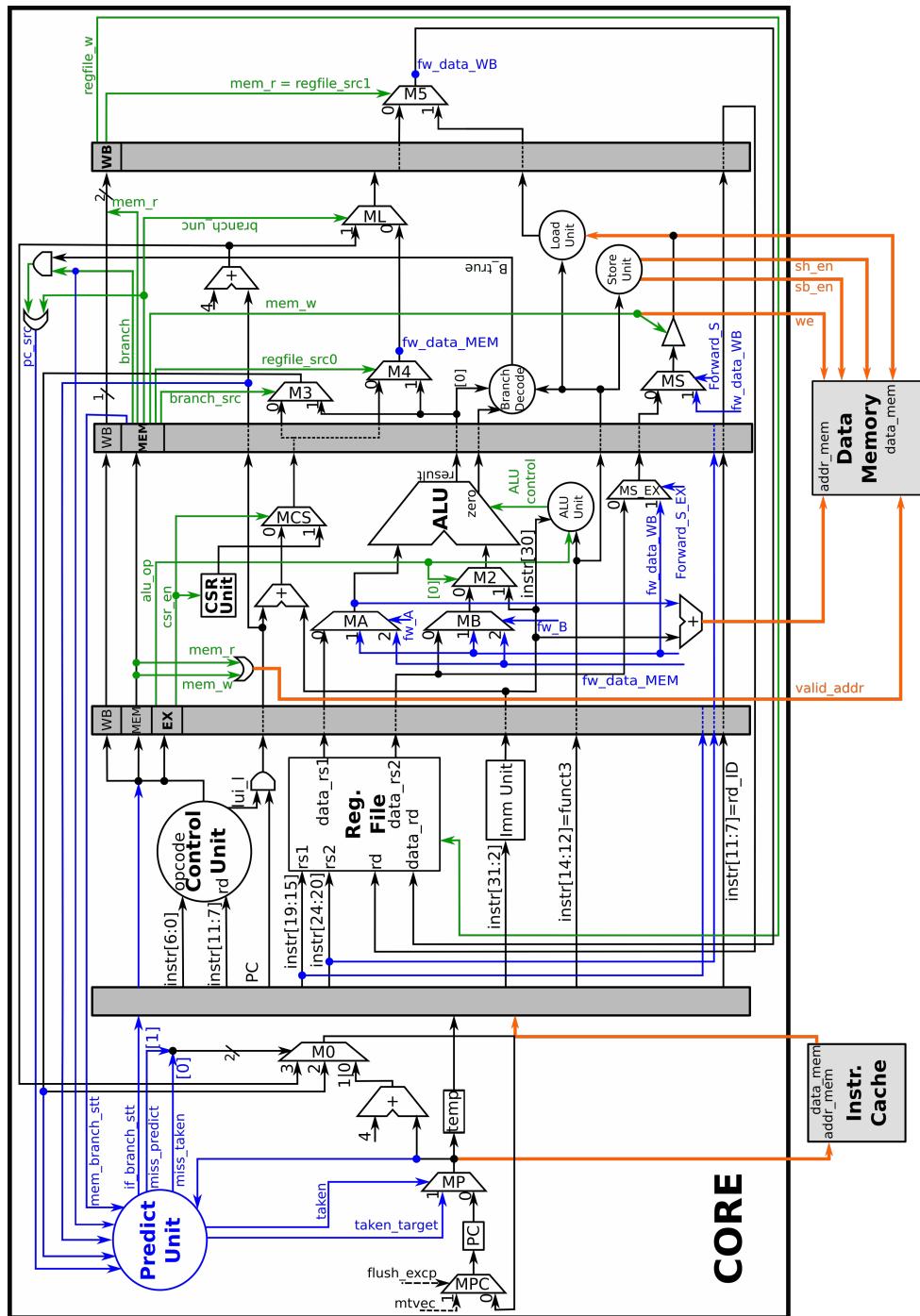
As adições de funcionalidade do núcleo se resumem a implementação dos registradores de controle, e da lógica de exceção/interrupção. As demais alterações realizadas tiveram como objetivo únicamente sua optimização, pois nessa etapa do projeto o foco era nos outros componentes necessários ao funcionamento do processador, como a memória cache. Porém é importante ressaltar a diferença que causou a mudança da base E para a I, e a remoção da extensão C na implementação do componente. A modificação da base apenas alterou o tamanho do banco de registradores, dobrando seu tamanho de 16x32 bits, para 32x32 bits.

O uso da extensão C havia sido projetada de forma que o núcleo não soubesse de sua existência, tal lógica era possível pelo fato de que as instruções da mesma eram decodificadas nas correspondentes da base E durante sua busca na memória. O decodificador em si era localizado na saída da memória, sendo assim a única consequência de sua remoção foi no incremento do contador de programa. Anteriormente era necessário saber se a instrução era da base E (32 bits longa) ou da extensão C (16 bits longa), para que se incrementa-se o PC de acordo, porém agora seu incremento é fixado em 4.

O novo caminho de dados, incluindo as modificações discutidas nas próximas subseções, pode ser visualizado na Figura 3.11. Evidencia-se a separação do barramento único que havia antes, em dois, um dedicado para busca de instruções, e outro para busca de dados. Todos os sinais relacionados ao PC, incluindo os endereços do preditor, foram reduzidos de 32 para 30 bits, pois o acesso as instruções é sempre alinhado em 4 bytes, os únicos locais em que se expande o PC para 32 bits é no estágio EX durante o cálculo de endereços alvos de desvio. Optou-se por deixar apenas esse momento com os 2 LSb para implementações de detecção de endereços inválidos, que podem gerar exceções. Também adicionou-se no estágio EX o bloco de CSR, que contém todos os registradores de controle do sistema, discutido na Seção 3.2.4, optando-se por não demonstrar suas conexões no mesmo diagrama para não dificultar o entendimento do núcleo.

Após a soma de PC com o imediato no estágio EX, os 2 LSb já são descartados, e nem são enviados ao estágio seguinte, ou seja, a lógica para detecção de endereços inválidos é implementada por meio de uma simples porta OR entre os bits [1:0], utilizada ainda no estágio EX. Endereços alvo de JALR possuem os 2 LSb, pois são resultantes da saída da ALU, podendo ser verificados no estágio MEM. O multiplexador MPC define a entrada de PC como o endereço alvo para tratamento de interrupções/exceções, sendo esse valor contido em um CSR no estágio EX, e escolhido na ocorrência de um flush por exceção (flush_excp).

Figura 3.11 – Caminho de dados do núcleo, estão ocultadas as unidades de forward, e detecção de load-stall do estágio ID.



Fonte: Autor.

3.2.1 Busca de Instrução

O maior causador de queda de desempenho no projeto era o acesso a memória. Constituída por blocos de RAM, elas apresentam uma latência mínima de 1 ciclo de relógio para leitura, sendo assim, a cada instrução buscada se inseria um NOP no pipeline enquanto a esperava. A fim de se remover essa espera de 1 ciclo, se reorganizou o acesso a memória. Ao endereçar a memória no estágio IF é armazenado também o valor do PC em um registrador temporário (temp), no próximo ciclo se envia o dado disponível no barramento em conjunto com o do registrador temporário (que corresponde ao PC da instrução no barramento), para o registrador IF/ID. Nesse mesmo ciclo o próximo valor de PC já está lendo um novo dado da memória, que fica disponível no próximo ciclo, e assim suscetivamente, eliminando a latência de 1 ciclo de relógio.

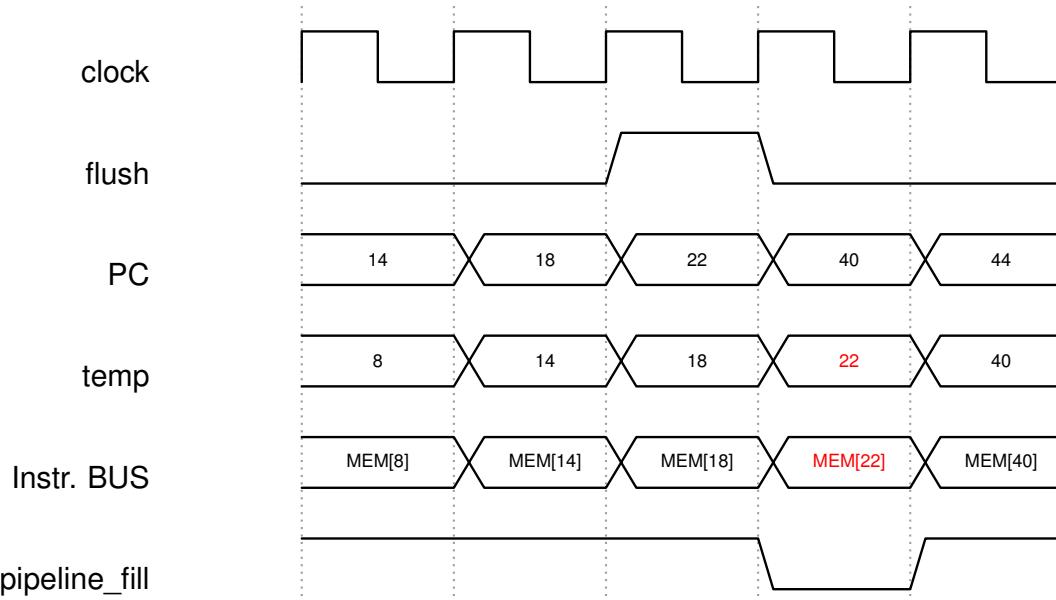
O único caso em que se espera pela memória é quando o pipeline está vazio, caso possível em um reset ou flush do núcleo. No primeiro ciclo após uma dessas duas ocorrências o barramento de instr. não contém um dado válido, e por isso deve ser ignorado, enviando-se assim por 1 ciclo a instr. NOP para o estágio 2. Durante o reset ou flush também é indicado a memória que o endereço no barramento não é válido, evitando assim a possível ocorrência de um miss na cache de instrução, e consequente acesso a memória principal por um dado que não se deseja.

O diagrama de tempo da Figura 3.12 apresenta o funcionamento da busca de instrução sem a latência, no primeiro ciclo se acessa a memória no endereço 14, enquanto se lê o barramento (instr. BUS) com o dado do endereço 8 acessado no ciclo anterior, esse mesmo endereço está disponível no registrador temp. No próximo ciclo então se tem o temp atualizado com o valor 14, e o dado no barramento agora correspondendo ao do endereço 14, enquanto o novo valor de PC, 18, já está acessando o próximo endereço. No terceiro ciclo ocorre um flush, alterando o valor de PC para 40, sendo assim, no quarto ciclo o dado disponível no barramento deve ser ignorado, pois a instrução do endereço 18 da memória foi descartada do pipeline. É então inserido um NOP no estágio 2 enquanto se espera pela instrução do novo endereço provido pelo desvio condicional/incondicional, sendo após um ciclo então voltado a se buscar uma instrução por ciclo.

Para que o endereço alvo do predictor não cause a perda de um ciclo em casos taken, pois necessitaria-se que o mesmo fosse primeiramente armazenado no PC e depois indexado seu valor na memória, se conectou ele diretamente ao barramento de endereço de instrução por meio de um multiplexador, MP. Dessa forma, quando se tem um branch taken ele já busca a instr. correta da memória, e armazena o valor do endereço alvo de desvio no registrador temporário de PC. Também incrementa seu valor e o armazena no PC.

O sinal pipeline_fill é uma flag gerada no estágio IF, que em nível 1 indica que o estágio está preenchido com uma instrução, sendo propagado ao longo do pipeline até

Figura 3.12 – Diagrama de tempo da busca de instruções.



Fonte: Autor.

o estágio MEM. Seu principal uso é para evitar a detecção de exceção/interrupção no estágio MEM logo após um flush do pipeline, pois o PC de retorno do tratamento é obtido do registrador EX/MEM, que após o flush possuí dados não válidos.

3.2.2 Busca de Dados

A busca de dados da memória, assim como na de instruções, sofria de uma latência de 1 ciclo, a fim de se compensar essa perda de desempenho adiantou-se o acesso. Antes se calculava o endereço alvo pela ALU, registrava-se o valor no registrador de pipeline EX/MEM, e então acessava-se a memória, travando o pipeline por 1 ciclo nessa etapa enquanto se esperava pelo resultado. Agora alocou-se um somador dedicado para o endereço alvo, adicionado o valor base lido do banco de regs. no ciclo anterior, ao sinal imediato. Como é perceptível, agora o endereçamento passa por um somador e diversos multiplexadores, adicionando latência de propagação ao sinal, sendo necessário realizar diversas optimizações no núcleo a fim de equilibrar a mudança.

Uma das compensações realizadas foi no caminho do forward, relacionado também a busca de dados. Em instruções LH/LB[U] se buscava o dado da memória, e no último estágio do pipeline que se expandia o dado para 32 bits, de acordo com a necessidade (partindo do bit [16] ou [8], para LH e LB, respectivamente). Seu posicionamento no último estágio acrescentava níveis lógicos ao caminho do forward partindo do estágio WB, sendo movida então a unidade de load para o estágio MEM, ficando fora de qualquer caminho de

forward, que afetava anteriormente o somador dedicado de endereçamento da memória.

O multiplexador que enviava para o banco de registradores o valor de PC somado ao offset (+4) foi reposicionado no estágio MEM, após o ponto de forward, sendo nomeado de ML. Isso é possível pois o seu valor de saída apenas será o do somador em instruções de jump, nas quais o pipeline sofre flush, e por consequência não importa o forward. Por fim, é importante notar que apesar desse aumento de lógica no caminho de endereçamento da memória de dados (somador + forward), antes se utilizava um barramento único para instruções e dados, e por consequência o endereço do estágio MEM já passava por um multiplexador que escolhia entre ele e o endereço do estágio IF para assumir o barramento.

3.2.3 Detecção de Exceção/Interrupção

A detecção de exceções é realizada com os dados do registrador de pipeline EX-/MEM, no quarto estágio, como demonstra a Figura 3.13. Esses sinais foram criados no estágio EX e ID, todos ativos em 1. Uma flag para endereço inválido é atribuída para o cálculo de branch/jump, verificado por uma simples porta OR entre os 2 LSb. Porém para instruções de load/store a verificação precisa diferenciar entre acesso a word, half-word ou byte, pois um SW no endereço 1 deve causar uma exceção, já SB no mesmo endereço não. Para isso se utiliza o campo funct3, a fim de diferenciar o tipo de acesso a memória, enquanto se utiliza também os 2 LSb do endereço gerado, nomeado de data_addr.

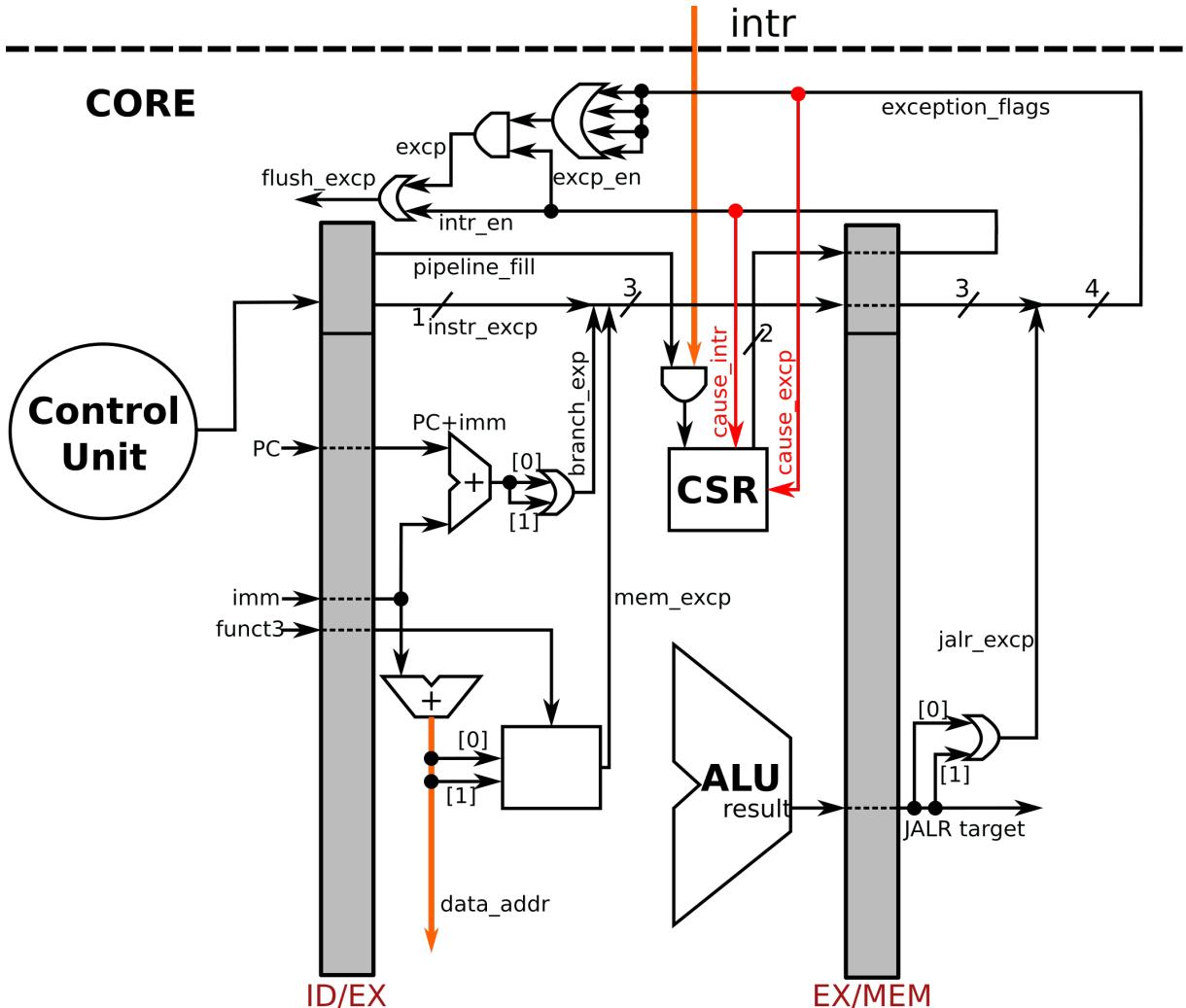
A detecção de instruções ilegais é realizada pela unidade de controle na decodificação, durante o estágio ID. como na base RV32I todas as instrução devem ter os 2 LSb em nível 1, se utiliza uma porta NAND para geração do sinal. A única detecção no estágio MEM é para instruções JALR, pois seu resultado são atribuídos da saída da ALU diretamente ao regs. de pipeline. Como um endereço de desvio gerado, sua detecção é da mesma forma que branch/jump no estágio EX. Na Tabela 3.1 contém os tipos de exceções/interrupções possíveis de serem detectadas.

Tabela 3.1 – Possíveis exceções/interrupções e suas causas.

Instrução / Origem	Causa
Exceção	
BEQ / BNE / BGE[U] / BLT[U] / JAL	addr_target[1:0] < "11"
JALR	jalr_target[1:0] < "11"
LW / SW	data_addr[1:0] < "11"
LH[U] / SH	data_addr[0] = "1"
Interrupção	
Interrupção externa	intr = "1"

Fonte: Autor.

Figura 3.13 – Detecção de exceções/interrupções.



Fonte: Autor.

A unidade dos registradores de controle envia para o estágio MEM em conjunto com as flags de exceções, o sinal de habilitação de exceção por software, contido no registrador mie, que é desabilitado caso o bit de exceção/interrupção global de mstatus esteja em nível 0. Caso o sinal seja 0, desabilita-se qualquer detecção por meio de uma porta AND, do contrário, gera o sinal flush_excp, que se distribui pelo pipeline da mesma forma que um flush ocorrido por jump. Com exceção de que agora se descarta também a instrução do estágio MEM, pois não se deseja o término da instrução que causou a exceção. O tipo de exceção é enviado para o registrador mcause, dentro da unidade CSR, para que auxílie na análise da exceção.

O pino de entrada de interrupção do núcleo é conectado a unidade CSR, e funciona como uma flag assim como as outras geradas, sendo colocado em nível 0 (inativo) pela unidade CSR no caso das interrupções estarem desabilitadas. Quando em nível 1, aproveita-se do mesmo caminho de execução de uma exceção, simplificando a lógica de

sua implementação, por mais que nesse caso não fosse necessário cancelar a operação de todo o pipeline. A interrupção tem prioridade sobre a ocorrência de uma exceção, sendo assim, no caso de ambas acontecerem no mesmo ciclo o tratamento que a unidade CSR irá gerar é para interrupções. A flag de interrupção é ignorada no estágio EX caso o pipeline não esteja preenchido até esse mesmo estágio (ocorrência de flush), pois o valor de PC contido no registrador EX/MEM, que seria armazenado no regs. mepc utilizado para retorno do tratamento não seria válido. O reconhecimento do preenchimento do pipeline é realizado pela flag `pipeline_fill`, criada no estágio IF e propagado até então, discutido na Seção 3.2.1.

É importante destacar que o núcleo não possui forma de reconhecer de qual dispositivo foi gerada a fonte de interrupção externa, pois é um único bit, sendo assim é necessário utilizar de polling por software para determinar qual dispositivo gerou o evento. Esse programa irá essencialmente ler o registrador de controle/status de cada dispositivo, a fim de descobrir se o mesmo gerou uma interrupção, sendo necessário estabelecer uma ordem de prioridade nessa verificação. No momento atual do projeto apenas uma fonte de interrupção externa é possível, a UART, porém a mesma pode gerar tanto interrupção por leitura (dado pronto para ser lido) quanto para escrita (pronta para enviar dado), conforme explicado na Seção 3.5.2. Sendo necessário diferenciar qual das duas fontes de interrupção da própria UART gerou o sinal, ou no caso das duas estarem ativa, verificar em ordem prioritária.

Durante o reconhecimento de uma exceção/interrupção se desabilita o sinal MIEN (interrupção global) do registrador mstatus, sendo restaurado apenas após a execução da instr. MRET, que finaliza o tratamento. Logo é responsabilidade do programador reabilitar o bit no registrador mstatus por meio de instr. CSR, caso se deseje a recepção de outros eventos durante o tratamento atual. Outras características relacionadas a exceções/interrupções, como a forma da unidade CSR reconhecer a causa do evento, são discutidas na Seção 3.2.4.

3.2.4 Registradores de Controle/Status

Implementou-se as instruções de acesso a registradores de controle/status do núcleo, CSR (Control/Status Register), que definem o estado atual de operação do hardware, para um dado nível de privilégio. A arquitetura atual conta apenas com o privilégio de M (Machine), e como não se tem os níveis S e U, diversos campos dos registradores utilizados podem ser ignorados. A Tabela 3.2 contém os registradores implementados, com suas respectivas finalidades.

O diagrama da Figura 3.14 apresenta a lógica da execução das instruções CSR, ocultando os demais componentes que não são relevantes. A fim de não se adicionar

Tabela 3.2 – Registradores de controle/status implementados.

Registrador	Função
mstatus	registrador de status
mie	habilitação de interrupção/exceção
mtvec	end. base da exceção/interrupção
mepc	PC da exceção/interrupção
mcause	causa da exceção/interrupção

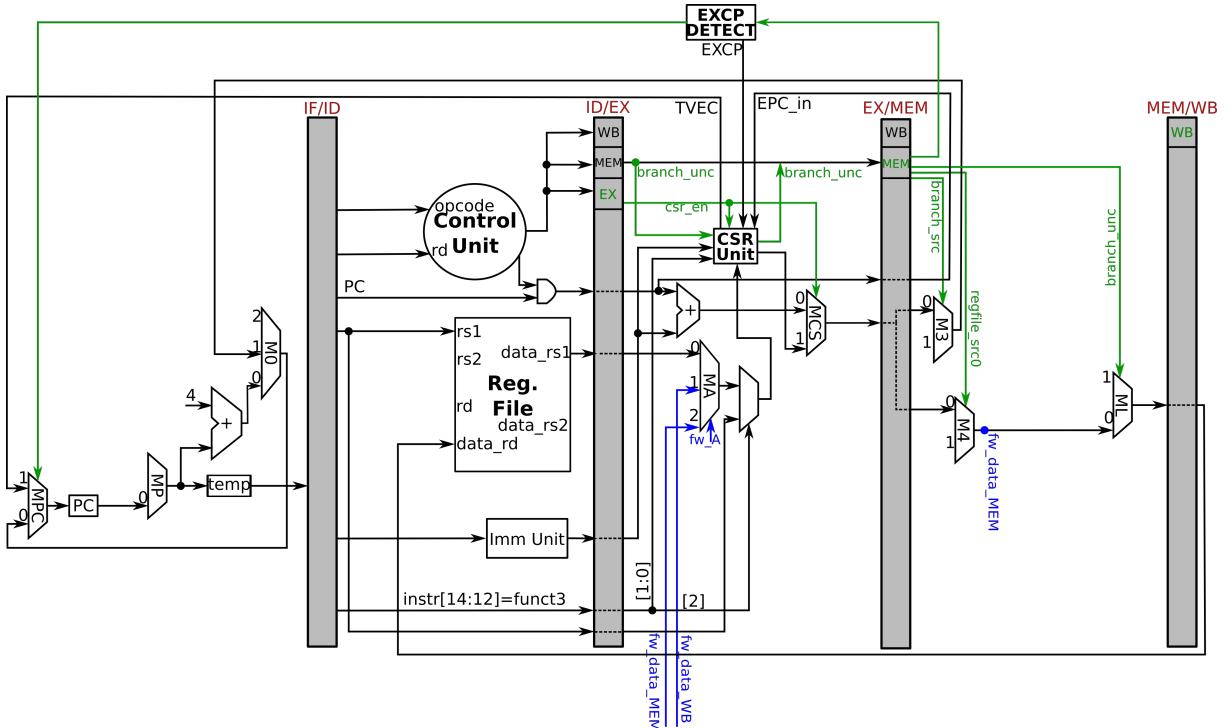
Fonte: Autor.

muita lógica ao núcleo, aproveitou-se de boa parte do caminho já utilizado em instruções de desvio incondicional, jump. O acesso a unidade CSR começa com os campos imediato e funct3, sendo o MSb de funct3 responsável por definir o dado de entrada de CSR. Sua escolha é entre o campo rs1 da instrução estendido para 32 bits (com 0s) ou o valor contido em rs1 no banco, que corresponde a saída do MUX A do caminho do forward para evitar dependências. Os 2 LSb de funct3 codificam a operação da instrução CSR, em que 0 se escreve o dado inteiro da entrada no regs. selecionado, enviando seu valor antigo ao registrador EX/MEM, que no final do estágio WB é salvo no banco. O valor 1 é para a operação de set bit, que coloca em nível 1 cada campo do regs. de controle, se o mesmo campo do dado de entrada estiver em 1, funcionando como uma máscara. Para o código 2 se executa a mesma lógica que o set bit, porém deixando em nível 0 e não 1 os campos correspondentes, operação clear bit.

A saída da unidade CSR é conectada ao multiplexador MCS, que é controlado por csr_en, colocado em nível 1 apenas em instruções CSR, o objetivo foi aproveitar o caminho de escrita no banco que o somador não ocultado do estágio EX já utilizava. Mais especificamente, a saída desse somador é salva no banco na execução de instruções LUI e AUIPC. No estágio MEM os sinais que controlam os restantes dos multiplexadores do caminho do banco funcionam de acordo, pois só são ativos (sinal de controle em nível 1) em instr. de desvio incondicional. Demonstra-se também a saída TVEC da unidade CSR, que corresponde ao valor do registrador mtvec, contendo o endereço base do desvio para tratamento de exceções/interrupções, conectado a um multiplexador adicionado na porta de entrada do PC. Também se tem como entrada em CSR o valor de PC no estágio MEM, armazenado no registrador mepc, para que se possa retornar após o término do tratamento por meio da instrução dedicada MRET.

Uma instrução de controle especial implementada é a MRET, que retorna de rotinas de tratamento, seu uso tem como alvo de escrita o registrador x0 (constante 0), logo pode ser ignorado o caminho do banco de registradores, e o registrador selecionado pela instr. na unidade CSR é o mepc, enviando ele então pelo MUX MCS, e pelo M3, aproveitando-se do caminho de branch/jump. Para isso, é necessário colocar em nível 1 o sinal branch_unc no estágio EX, apenas para a execução de MRET, no restante dos casos

Figura 3.14 – Execução de instruções CSR.



Fonte: Autor.

o sinal branch_unc permanece com seu valor inalterado na passagem pela unidade CSR. A instrução de retorno também reabilita o bit de interrupção/exceção global, contido no registrador mstatus, que havia sido desabilitado durante a recepção do evento.

3.2.4.1 mstatus

O registrador de status possuí apenas 2 bits válidos, o de habilitação de interrupções/exceções global para nível privilegiado M, MIE, e o MPIE, que armazena o valor anterior de MIE na ocorrência de uma exceção/interrupção. O campo MIE está posicionado no bit [3] do registrador, e o MPIE no [7], os demais são conectados diretamente ao GND, sendo qualquer escrita ignorada, e sua leitura retornado o valor 0. Quando uma exceção ocorre, o bit [3] é colocado em nível 0, desabilitando as interrupções até o retorno da função de tratamento por meio da instrução MRET, ou escrita manual do bit por uma instrução CSR por software. O bit [3] como uma flag global de interrupção desabilita as flags individuais, definidas no registrador mie, sendo utilizada no estágio EX e MEM, pois em ambos se tem a possível ocorrência de exceções. Interrupções são detectadas ainda no estágio EX, e ignoradas quando o campo MIE está em nível 0, ou então o campo [11] do registrador mie, de habilitação individual de interrupções/exceções.

3.2.4.2 *mie*

O registrador de habilitação de interrupção/exceção controla o uso individual dos tratamentos, possibilitando apenas a habilitação de interrupções externas, ou exceções por software, por exemplo. Para isso, o bit de interrupção/exceção global do registrador mstatus deve estar habilitado, do contrário as flags desse registrador serão desconsideradas. Dentre os diversos campos contidos nele, se implementou os bits [11] e [3], correspondendo a habilitação de interrupção externa e exceções por software, respectivamente. O restante dos campos quando lidos devolvem o valor 0, sendo sua escrita ignorada.

3.2.4.3 *mtvec*

O registrador de vetor armazena o endereço de desvio na ocorrência de uma exceção/interrupção. Sua saída está ligada diretamente a um multiplexador, M_excp, que define a entrada de PC. Seu valor deve ser alinhado em 4 bytes, logo apenas os bits [31:2] são utilizados no desvio, os 2 LSb definem o modo de operação, sendo implementado o DIRECT e VECTOR. No DIRECT, o endereço de desvio consiste únicamente no armazenado no registrador, para qualquer situação, no modo VECTOR, o endereço desvio será o base somado ao valor do registrador mcause, porém apenas em interrupções. Em exceções, continua sendo apenas o endereço base. Na inicialização da FPGA, seu valor é colocado em 0, logo é responsabilidade do programa (preferencialmente um bootloader) ajustar seu valor para o inicio da tabela de vetores de exceções/interrupção.

3.2.4.4 *mepc*

O registrador mepc serve para armazenar o valor do endereço da instrução causadora da exceção, que não deve ser finalizada, sendo concluída apenas no retorno do tratamento. Seu valor é de 30 bits, pois não deve conter endereços inválidos, sendo alinhado de 4 em 4 bytes. Sua entrada está conectada ao PC propagado até o estágio MEM, pois é nesse estágio que se detecta a ocorrência de exceção. Vale ressaltar que em interrupções não se tem a necessidade de impedir a execução da instrução, porém, a fim de simplificar o controle, a mesma é cancelada, da mesma forma que na exceção.

3.2.4.5 mcause

O registrador mcause armazena um código que define a causa da exceção/interrupção, sendo o MSb colocado em nível 1 para indicar uma interrupção, e 0 para exceção. Nos 4 LSb se encontra o código da causa, que é somado ao valor base de mtvec em interrupções, quando o mesmo opera no modo VECTOR. A Tabela 3.3 contém a codificação da causa armazenada no registrador, aonde apenas alguns dos possíveis valores foram utilizados, além disso, os bits [30:4] são conectados diretamente ao GND.

Tabela 3.3 – Tabela de causas de exceção/interrupção.

MSB[31]	Cause[3:0]	Tipo
0	0000	end. de instr. desalinhado
0	0010	instr. ilegal
0	0100	end. de dado desalinhado
1	1011	interrupção externa

Fonte: Autor.

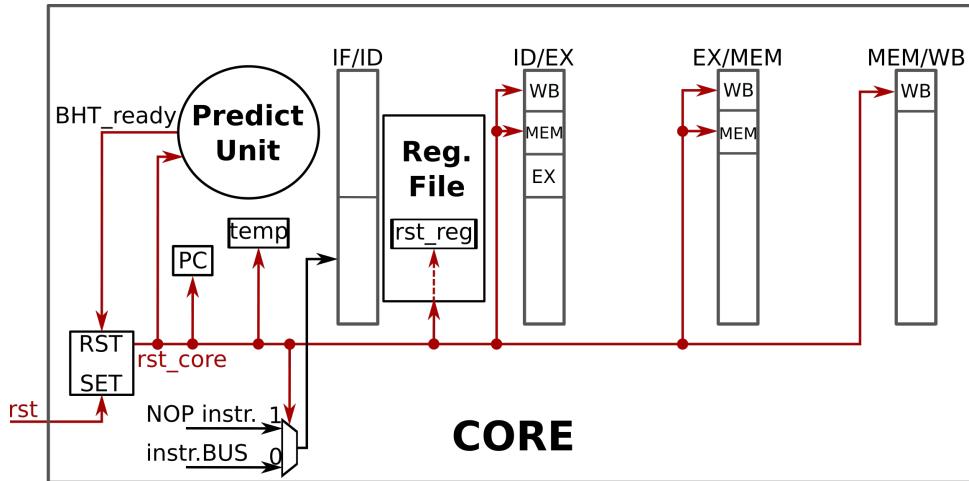
3.2.5 Lógica de Reset

O uso de reset no núcleo foi uma das partes mais otimizadas, o seu peso na latência do núcleo e no número de conexões por necessitar da capacidade de colocar diversos flip-flops em um estado conhecido foi repensado a fim de utiliza-lo apenas no essencial. Os registradores de pipeline por exemplo, agora colocam em nível conhecido (0) os sinais de controle, operadores passados de estágio em estágio são deixados no estado atual, apenas no estágio IF que se tem a necessidade de passar a codificação de uma instrução completa, a NOP. A Figura 3.15 apresenta o reset no pipeline, e os elementos que são afetados por ele. Na realização de um reset, o PC é ajustado para o valor 0, apontando para o ínicio da seção do bootloader, a fim de se carregar um programa.

Outra alteração realizada foi a aplicação do conceito de pipeline na recepção do reset, anteriormente era necessário considerar a latência do reset desde o pino de entrada da FPGA, até cada FF do núcleo. Agora o mesmo é armazenado em um FF na entrada do núcleo, e a partir dessa localização, se distribui aos elementos necessários, diminuindo a influência na frequência de operação máxima do núcleo. No estágio 2 se coloca em nível 0 todos os FF do registrador de reset de 32 bits do banco de registradores, que teve sua lógica de reset reprojetada conforme discutido na Seção 3.2.6.

No estágio 1 se coloca em 0 os dois registradores que armazenam informações referentes ao endereço de acesso a memória para busca de instruções, PC e seu registrador temporário. Também se envia o sinal de reset para o preditor de desvios, que deve limpar

Figura 3.15 – Lógica de reset do núcleo.



Fonte: Autor.

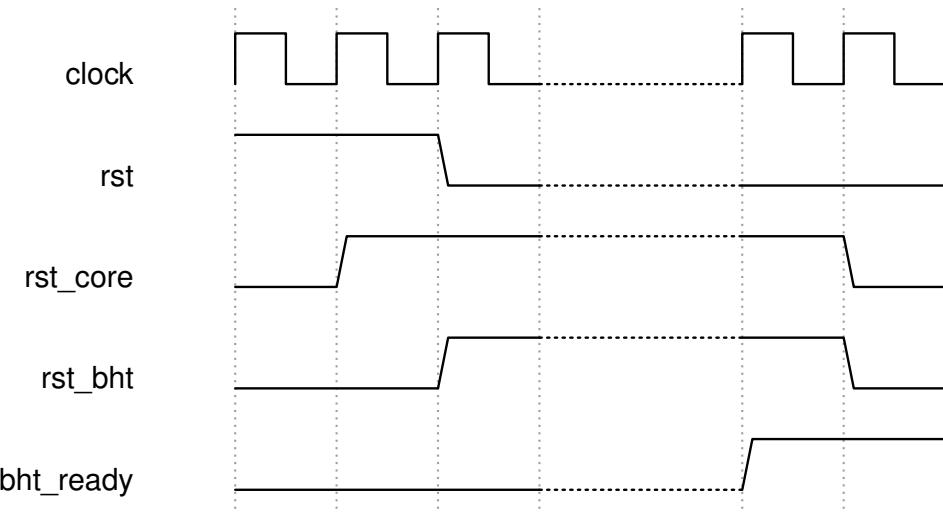
os bits de validade da BHT, lógica essa apresentada na Seção 3.2.8. Como o preditor não fica pronto em um único ciclo como os demais FF do núcleo devido a seu grande armazenamento de memória interno, o mesmo indica por uma flag quando está pronto, `bht_ready`. Quando ela está em nível 1, é colocado em 0 o sinal `rst_core`, indicando que o núcleo está pronto. O uso da flag do preditor como condição de término do reset é possível devido ao fato de que ele é o único componente que leva mais de 1 ciclo para ficar pronto.

O tempo total em que o componente fica em estado ocioso, é de 1 ciclo para colocar em nível 1 o FF de reset da entrada do núcleo somado mais um ciclo para que a BHT comece o processo, adicionado-se o total necessário para limpar a BHT, 512 ciclos, por fim somando-se 1 extra para reconhecer o término da BHT. No diagrama de tempo da Figura 3.16 se demonstra o reconhecimento do reset do sistema por parte do núcleo, sua entrada nesse processo, e saída ao receber a flag `bht_ready` em nível 1. O sinal `rst_bht` é uma flag utilizada apenas para demonstrar quando a BHT começa a limpar seu array de bits de validade, sendo assim, a área pontilhada horizontalmente do diagrama representa um total de 511 ciclos, o primeiro já está representando. O sinal `bht_ready` permanece em nível 1 até que se escreva algum dado na BHT, por consequência se um reset ocorrer antes dessa escrita, o processo de reset do núcleo será realizado em um único ciclo, em que se verifica o nível de `bht_ready`.

3.2.6 Banco de Registradores

Com a mudança da base E para a I do RISC-V, se aumentou o banco de 16 registradores de 32 bits, para um total de 32, com o registrador `x0` sendo ainda a constante 0. Essa decisão considerou que uma das principais características de dispositivos FPGA

Figura 3.16 – Diagrama de tempo da operação de reset do núcleo.



Fonte: Autor.

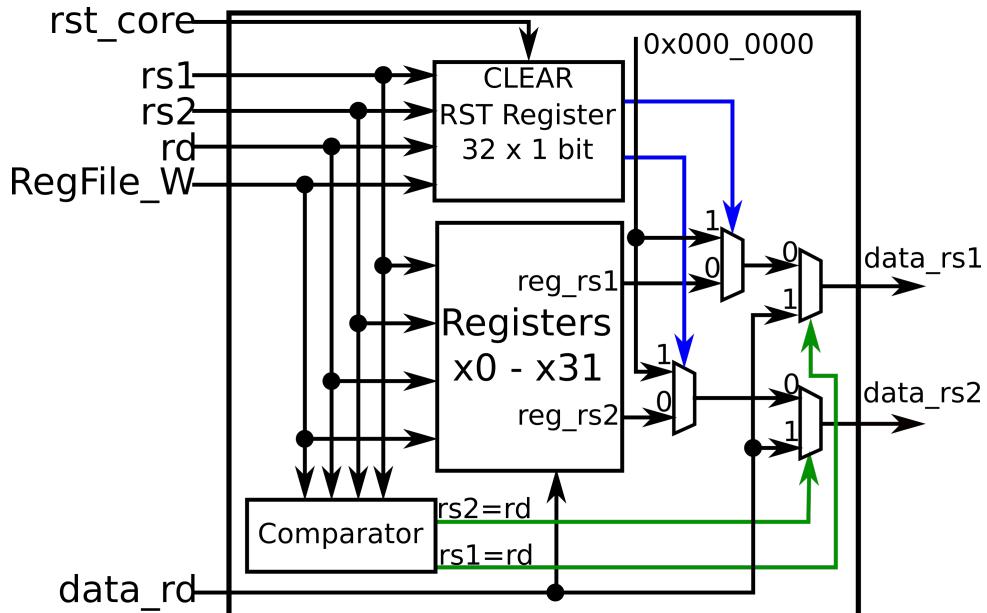
consiste no elevado número de FFs disponíveis, sendo o acréscimo mais significativo de hardware nos multiplexadores extras adicionados na seleção de entrada/saída de dados do componente de memória. O banco de registradores teve sua lógica refeita a fim de mudar sua implementação para uma com uso de RAM distribuída da FPGA Spartan-6, e não puramente por lógica reconfigurável. O uso de blocos de RAM já havia sido descartado anteriormente pela dificuldade com o número de portas, pois um BRAM tem no máximo 2 portas, e necessita-se de 2 para leitura e 1 para escrita, e também devido ao fato de um bloco de RAM proporcionar muito mais armazenamento do que o necessário, resultando em desperdício.

A fim de se utilizar RAM distribuída se tornou necessária a remoção do reset geral do bloco. Essa funcionalidade foi alcançada ao limitar o reset a um único registrador de 32 bits, que contém uma flag indicando a validade de cada registrador do banco. Quando acessamos um registrador nos endereços rs1/rs2, também verificamos a validade no bit correspondente do registrador de reset, caso o dado não seja válido, a saída do endereço correspondente será multiplexada para o valor 0. Em paralelo ocorre a verificação do forward interno, caso um endereço de escrita corresponda a um de leitura no ciclo atual, o dado novo será multiplexado para a saída, sendo essa ação de uma prioridade superior a de multiplexação pelo bit de validade. Sempre que ocorrer uma escrita no banco de registradores, o bit correspondente ao endereço a ser alterado no banco, é colocado em nível 1.

Como o registrador x0 é a constante 0, tanto o bit [0] do registrador de reset quanto o registrador x0 do banco são desconectados, pois nunca são escritos, sendo as conexões que os utilizam conectadas diretamente ao GND. A Figura 3.17 demonstra a lógica de leitura/escrita, com a verificação de reset e forward do banco de registradores. O sinal

de reset é nomeado de `rst_core` por agir apenas dentro do núcleo, seu funcionamento é elaborado na Seção 3.2.5.

Figura 3.17 – Banco de registradores.



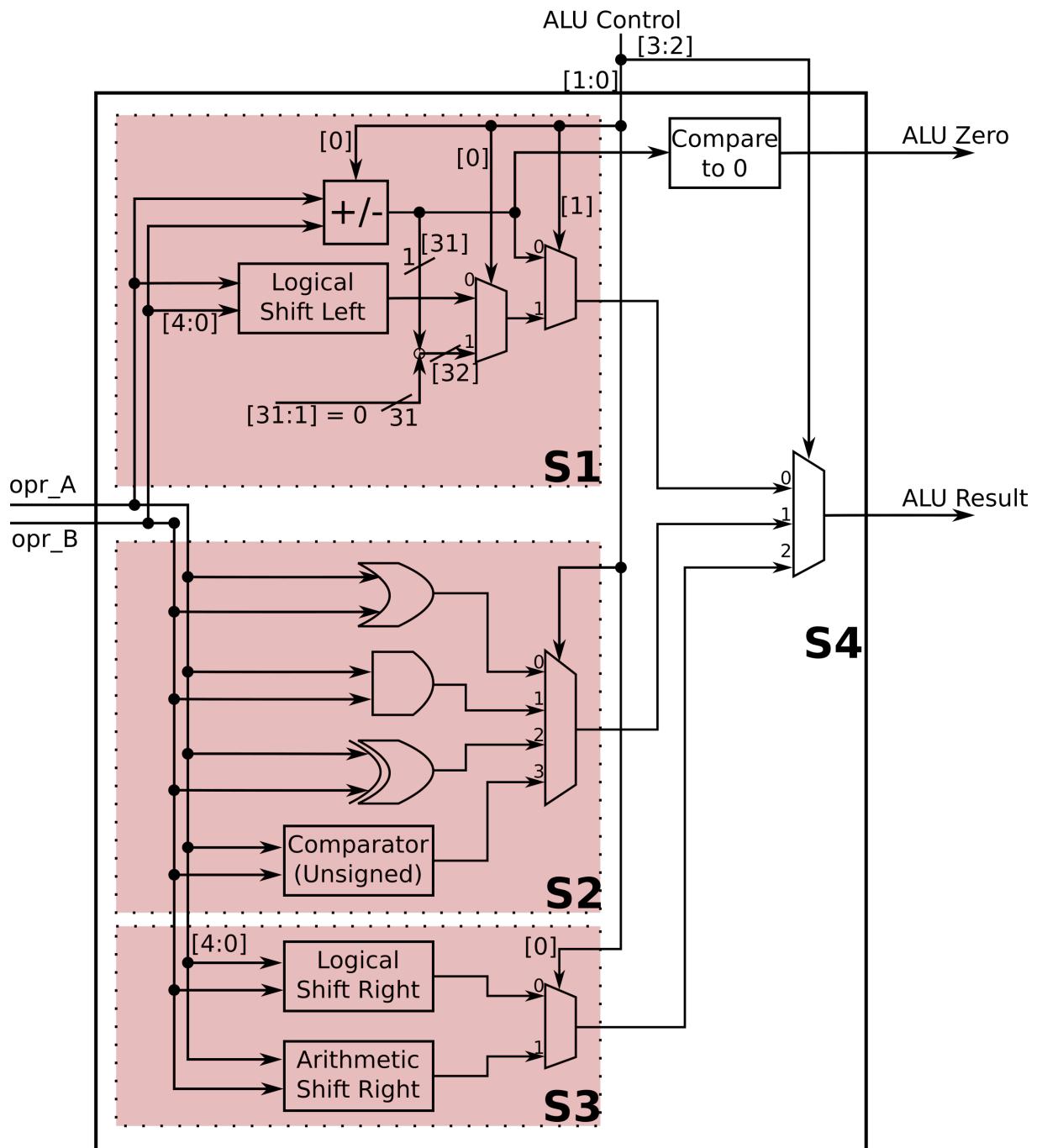
Fonte: Autor.

3.2.7 Unidade Lógica e Aritmética

As modificações realizadas na ALU visaram unicamente a redução da latência na sua execução, que estava com grande carga, principalmente no caminho que envolvia o uso da técnica de forward, passando pela operação de deslocamento. Como não se adicionou nenhuma extensão, as operações necessárias da ALU continuam as mesmas, porém alguns blocos foram otimizados (ou até removidos), reduzindo ainda mais o tempo de execução do componente. A Figura 3.18 contém um diagrama de blocos interno do componente, em que se demonstra cada operação realizada, divida em quatro seções.

Na seção S1 se define entre as operações de soma/subtração, deslocamento lógico para a esquerda, ou então SLT (set less than) com sinal. Anteriormente o uso de SLT era por meio de um comparador dedicado para operadores com sinal, porém agora se utiliza o MSB da subtração para indicar o resultado. Para isso, a codificação deve, selecionar simultaneamente a operação de subtração, e multiplexar para a saída da seção o resultado de SLT. A Tabela 3.4 contém a nova codificação utilizada nas operações do componente, aonde se percebe que o LSB do sinal de controle que define a operação de subtração é o mesmo para SLT. Essa codificação (SLT) também é utilizada nas instruções BEQ e BNE (branch if equal / not equal), em que se aproveita o resultado da subtração, e o compara a

Figura 3.18 – Lógica interna da ALU, com as seções S1, S2, S3 e S4.



Fonte: Autor.

constante 0 na seção S4, resultando na saída ALU Zero. A seção S1 é controlada pelos bits [1:0] do sinal de controle, ALU Control.

Tabela 3.4 – Codificação das operações realizadas pela ALU.

ALU Control	Operação
0000	+
0001	-
0010	Set Less Than
0011	Logical Shift Left
0100	OR
0101	AND
0110	XOR
0111	Set Less Than Unsigned
1000	Logical Shift Right
1001	Arithmetic Shift Right

Fonte: Autor.

A seção S2 controla as operações lógicas OR, AND e XOR, e também o uso de SLT Unsigned, que por sua vez continua utilizando um comparador dedicado para operadores sem sinal. Assim como na seção S1, seu controle é feito pelos bits [1:0] da entrada de controle. A seção S3 é dedicada a operações de deslocamento a direita, lógica e aritmética, sendo definida unicamente pelo bit [0] de ALU Control. Por fim, na seção S4 se multiplexa para a saída a operação desejada, de acordo com os bits [3:2] de ALU Control.

Essa nova organização possibilitou também a simplificação da codificação das operações. Um ponto a ressaltar é a remoção da operação de transferência do operador B para a saída. Essa função era utilizada com a instrução LUI, que agora aproveita o somador dedicado de PC com o imediato, no estágio EX, para realizar a mesma tarefa. Para isso, o valor de PC deve ser colocado em nível 0, pois o imediato de LUI constitui por si só todo o resultado desejado. Essa detecção (e consequente ajuste do PC para 0) é realizada no estágio anterior, ID, que não acrescenta latência ao caminho dados, pois a transferência do PC entre o registrador de pipeline IF/ID para ID/EX era realizada anteriormente sem nenhuma lógica em seu caminho.

3.2.8 Predito de Desvios

Os acréscimos do preditor se resumem a lógica de reset, sendo assim suas características continuam sendo a de uma memória com mapeamento direto de 512 linhas. Em cada linha se armazena um endereço alvo de desvio, uma tag e 1 bit de validade para verificação, e 2 bits de predição, pois o sistema preditor utiliza a máquina de estados finitos de 2 bits. Removeu os 2 LSB de todos os sinais envolvendo o PC, já que eles não

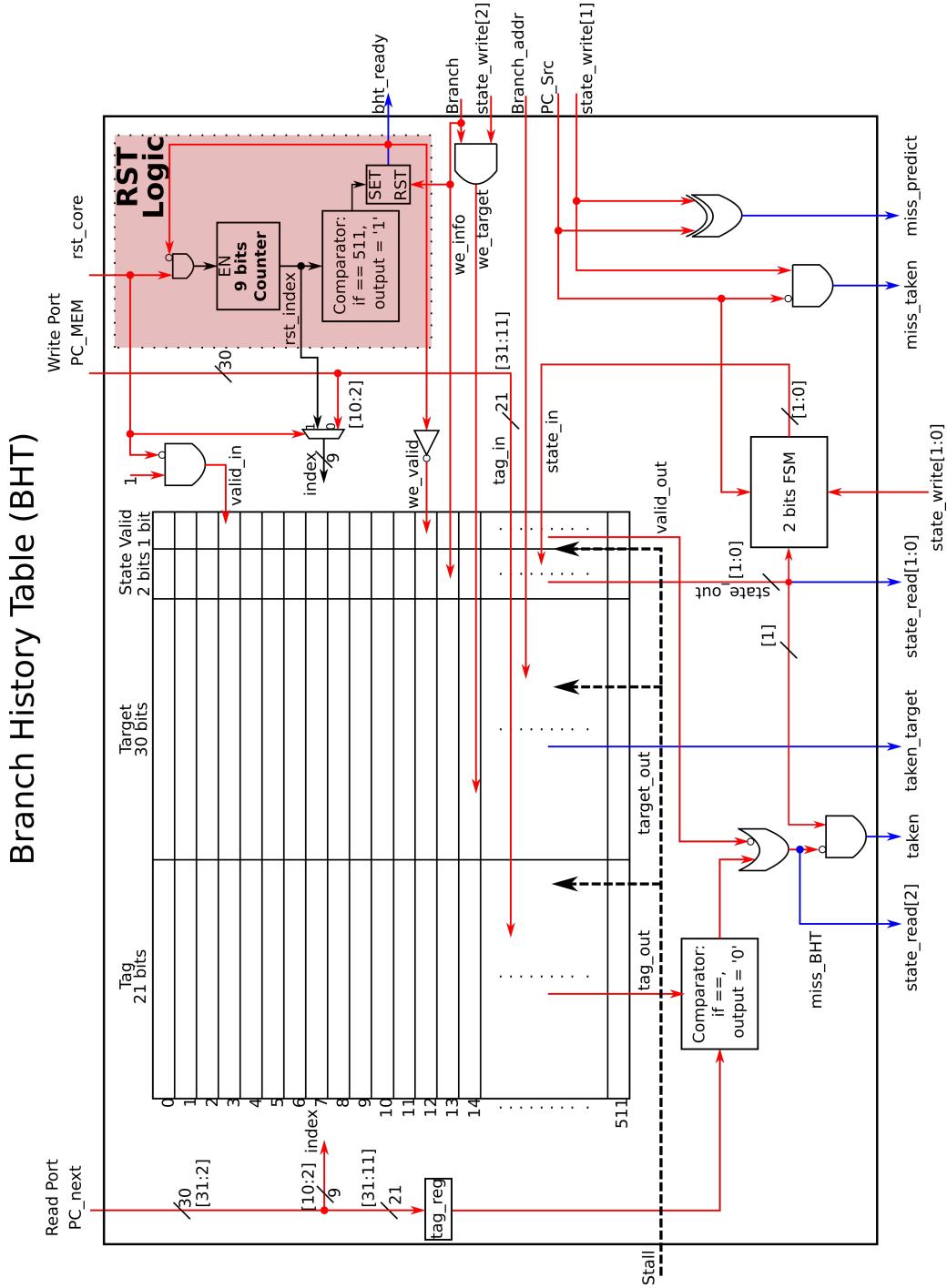
acrescentam nenhuma informação devido ao alinhamento de 4 bytes das instruções, lembrando que a extensão C (que proporciona instruções de 2 bytes, e consequentemente causa um relaxamento no alinhamento) foi removida. O sistema de memória completo, a BHT (Branch History Table) ocupa um total de 2 BRAMs da FPGA Spartan-6, sendo separado um bloco de RAM para o endereço alvo de 30 bits (como dito, se removeu os 2 LSB), e outro bloco de RAM para a tag e os 2 bits de estado. O array de bits de validade foi implementado por meio de memória distribuída. A Figura 3.19 apresenta o bloco preditor de desvios, com a BHT, e sua lógica de acesso para leitura e escrita, assim como a nova funcionalidade acrescenta, o sistema de reset.

Anteriormente, o preditor não limpava os dados atuais da tabela na ocorrência de um reset, sendo uma das implementações essências pendentes. Agora, na ocorrência de um reset do sistema especificado pelo sinal `rst_core`, a BHT irá passar por um processo que no qual se coloca em nível 0 todos os seus bits de validade. O contador de 9 bits (tamanho do ponteiro de endereçamento da BHT) tem seu estado inicial em 0, ao receber um reset do núcleo sua contagem é habilitada fazendo com que sua saída, `rst_index`, selecione por meio de um multiplexador controlado também pelo sinal de reset, cada uma das linhas da BHT até a última. O nível do bit de validade a ser escrito também é colocado em 0 pela porta AND com entrada inversora por parte do sinal `rst_core`. A habilitação de escrita é separada para cada um dos blocos de RAM, `we_target` para o endereço alvo, e `we_info` para o bloco formado pela tag com os 2 bits de validade e também a memória distribuída de validade. Um terceiro sinal de habilitação é o `we_valid`, que habilita a escrita únicamente da memória distribuída de validade durante o reset.

O processo completo leva 512 ciclos, o total de linhas da BHT para invalidar toda a memória. Quando o ponteiro utilizado para endereço de reset alcança o valor 512, um comparador realiza a função SET do flip-flop responsável por sinalizar o término da operação, através da flag `bht_ready`. A mesma continua em nível 1 (indicando que a BHT está totalmente inválida) até que uma operação de escrita ocorra, a colocando novamente em nível 0. Caso um novo reset ocorra, mas a BHT não tenha nenhum dado salvo ainda (`bht_ready = 1`), o contado não é habilitado, e a flag continua indicando que a BHT já está pronta para operar. Considerou-se utilizar os bits de validade como um grande registrador de 512 bits com reset geral em um ciclo único, e não como uma RAM distribuída sem essa função, porém o acréscimo de lógica dessa forma dispensou a ideia.

O sinal de Stall em nível 1 trava todas as saídas da BHT, impedindo que se perca o estado lido. Uma alteração na BHT a ser considerada seria o fato de que antes ao se realizar a leitura do estado, propagava-se internamente os 2 bits lidos por N ciclos, até o momento de verificação. Como a verificação ocorre no quarto estágio (MEM), se utilizava 3 registradores temporários para propagar o estado lido e verificar se o preditor havia acertado. Essencialmente, esses registradores temporários correspondem a mesma lógica que os de pipeline, porém localizados internamente no controle da BHT, porém

Figura 3.19 – Sistema predictor de desvios dinâmico.



Fonte: Autor.

optou-se por mover eles para fora, distribuindo-se então em conjunto com os outros sinais de controle. Essa alteração simplificou a lógica de flush e stall, pois como dito, eles são essencialmente regs. de pipeline, e devem sofrer desses eventos também. Os 2 bits de estado lidos correspondem ao sinal `state_read[1:0]`, e o valor resgatado no estágio MEM, `state_write[1:0]`. O MSB desse sinal completo, `state_read[2]`, é a flag que em nível 1 diz quando não se encontrou nenhuma correspondência na BHT.

Outras alterações pequenas incluem a adição de um regs. temporário para armazenar o PC de leitura, pois é necessário verificar no próximo ciclo (latência de acesso a BRAM) se a tag corresponde. Anteriormente não era necessário, já que o pipeline era travado por 1 ciclo a cada instrução acessada, no mesmo estágio de acesso a BHT. Também conectou-se o sinal `PC_Src` dedicadamente a unidade do preditor, não se utilizando mais em nenhum outro local. Essa simplificação é possível no momento em que se interpreta instruções de jump (que utilizavam esse sinal) como um branch sempre not-taken, e que não irá ser armazenado nas BRAMs. Logo, sua execução irá sempre gerar um erro de predição, e corrigir o PC com o endereço alvo de desvio, dando flush no pipeline.

3.3 MEMÓRIA CACHE

O projeto da memória cache considerou dois mapeamentos diferentes, direto e associativo por conjunto. Com o número de blocos de RAM disponível e suas possíveis configurações, o uso de uma memória associativa por conjunto com apenas 2 palavras se tornou inviável. Considerando-se que o desejado era de se obter uma cache de no mínimo 16 kB de instr. e 16 kB de dados.

A implementação ficou definida então como uma cache de 16 kB de instr. e 16 kB de dados, com mapeamento direto e blocos de uma palavra de 32 bits, ocupando-se 26 BRAMs no total. Com o seu tamanho definido, é possível associar que dos 32 bits do endereço de acesso, 20 são utilizados como tag e os 12 restantes para endereçar a cache. A divisão dos campos foi realizada de forma igual para ambas as caches, se utilizou uma memória para o armazenamento do dado a ser lido/escrito, e outra memória para armazenamento da tag em conjunto com um bit de validade da linha. Ressaltando que a cache de dados ainda possui um bit extra em conjunto com a tag e a validade, o bit de modificação, pois diferente das instr., seus dados podem ser modificados. A Tabela 3.5 apresenta a divisão dos campos da cache em diferentes memórias. A separação facilita o reset dos mesmos quando se deseja invalidar a cache, evitando também escritas desnecessárias nos dados, pois apenas o bit de validade precisa ser alterado.

Tabela 3.5 – Divisão dos campos da cache em diferentes memórias, mod representa o bit de modificação, ativo em nível 1.

	Instr. Cache	Data Cache
MEM 1	instrução	dado
MEM 2	validade+tag	validade+mod+tag

Fonte: Autor.

3.3.1 Coerência de Dados

A cache possuí a lógica de write-back, evitando a escrita na memória principal durante a ocorrência de cada store realizado. Em conjunto com o write-back, também se utiliza a política de write-allocate, na qual se busca o dado da memória principal sempre que um write-miss é gerado. Pois de acordo com o princípio de localidade temporal, ele tende a ser acessado novamente. Ainda sim, como o armazenamento de cada linha corresponde a exata uma palavra, não se tem a necessidade de buscar uma palavra da memória somente para ser sobre-escrita, desperdiçando tempo acessando a memória RAM.

Dividiu-se então a detecção de write-miss em miss_w e miss_hb, com miss_w correspondendo a detecção de miss em store word que é ignorado. A flag miss_hb indica a ocorrência de miss por store em half-word ou byte, que deve então gerar uma solicitação pela palavra de 32 bits correspondente a linha acessada, antes de escrever na cache. Não se implementou nenhuma lógica de coerência entre as próprias caches L1 de instrução e dados, desconsiderando-se a possibilidade da execução de programas que se modificam em tempo de execução.

3.3.2 Lógica de Reset

O reset é realizado por meio da escrita sequencial, partindo do endereço 0 até o último, colocando em nível 0 cada bit de validade da cache de instr. e dados. Como as cache possuem endereçamento a blocos de 1 palavra, com um total de 16 kB (o reset ocorre em paralelo na de instr. e de dados), temos 4096 endereços para acessar. O tempo mínimo total para se invalidar a cache é de 4096 ciclos de relógio, nesse intervalo seus sinais de controle com o núcleo (stall) estão ativos, para travar o componente a espera de dados. Também se desativa qualquer sinal de acesso a memória principal, para evitar a busca de dados indesejado. De fato, a lógica de reset da cache funciona da mesma forma que o reset da BHT no núcleo, descrito na Seção 3.2.8. Pois ambas as memórias são formadas por blocos de RAM, com endereços armazenando palavras de 32 bits e mapeamento direto, a única diferença está no tamanho total da memória, e por consequência o tempo utilizado invalidando a mesma.

3.3.3 Inicialização da Cache

Sendo possível inicializar os blocos de RAM da FPGA Spartan-6 com um valor previamente definido durante o momento de configuração da FPGA, optou-se por carregar o código do bootloader. O carregamento do bootloader na configuração do dispositivo é uma necessidade pelo fato de se utilizar apenas uma memória externa volátil, e nenhuma não volátil para armazenamento de programas. Com um tamanho de 16 kiB para a cache de instruções, estabeleceu-se que o tamanho do bootloader não deve passar disso. Após o carregamento do programa de usuário pela função do bootloader, o mesmo seria perdido pois sua inicialização direto na cache significa que o código não está presente na memória principal, e a cache de instrução não escreve na memória.

Dessa forma seria necessário reconfigurar a FPGA toda vez que se desejasse enviar um programa para o processador. A fim de evitar isso, o programa do bootloader é carregado tanto na cache de instrução, quanto na de dados, sendo que na de dados é realizado no final do programa de boot então uma operação de write-back, armazenando o código na memória principal. Possibilitando assim o carregamento de diferentes programas de usuário para a memória sem a necessidade de ficar reprogramando a FPGA. É importante destacar que para carregar o bootloader em ambas as caches ele deve conter somente a seção de texto, para evitar a sobreposição de instruções e dados em uma mesma linha. Tarefa relativamente simples, já que suas ações consistem em apenas carregar valores para registradores de controle, e configurar os dispositivos I/O. A única função que exige relativo cuidado na escrita foi a que recebe o código do usuário pela UART, e o salva na memória. Conforme explicado na Seção 3.2.5, após um reset do sistema o PC volta para o endereço 0, início da seção do bootloader, para carregamento de um programa novo.

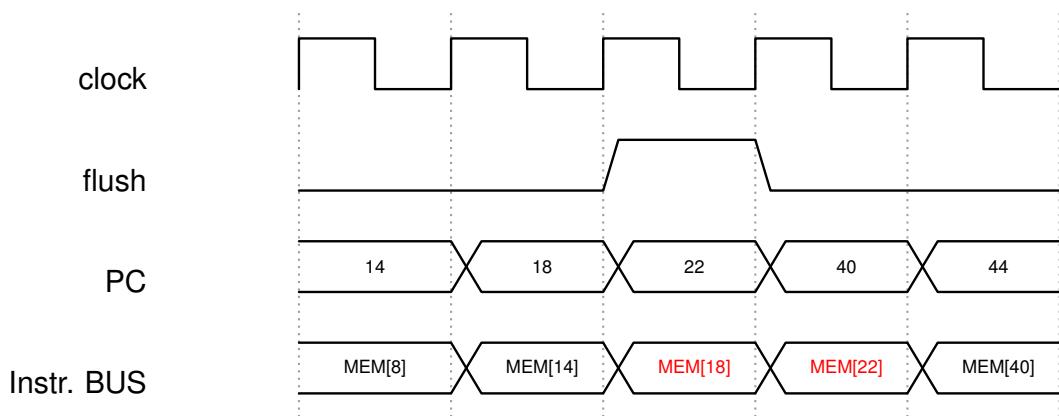
3.3.4 Cache de Instrução

A cache de instrução possuí um funcionamento relativamente simples, obtido em grande parte devido a mesma estar sempre sendo acessada no estágio IF do núcleo, e não possuir a operação de escrita. O fato dela estar sempre sendo acessada no estágio IF elimina a necessidade de adicionar um estado ocioso a mesma, aonde se ignora o endereço de entrada da sua porta, simplificando seu controle. A única necessidade em que se para o acesso a ela é quando a cache de dados sofre um miss, pois seu acesso é mais adiante no pipeline, sendo necessário travar a cache de instrução. Essa troca de informação (stall pela cache de dados) é realizada internamente pelo controle geral da cache, evitando a formação de um caminho que vai da cache de dados para o núcleo, e retorna a cache de instrução.

Além do stall que a cache de instr. pode sofrer para não perder o dado em sua porta enquanto espera pela cache de dados, é importante ressaltar o efeito de um flush no núcleo. Na ocorrência de um flush o sinal de validade do endereço da cache de instr. será colocado em nível inativo. O dado na porta é ignorado de qualquer forma, mas se tem a necessidade de indicar a inválidação do ciclo de leitura para que a mesma não gere um miss de uma instrução que não deveria ser buscada. Enquanto o bit de validade estiver indicando um ciclo inativo de leitura, a cache não irá gerar miss, e caso um dado já esteja sendo buscado da memória principal para a cache, será cancelada a operação.

Os casos em que se tem o bit de validade inativo é na ocorrência de flush, mais especificamente, no ciclo em que é realizado o branch, e no próximo ciclo, pois o pipeline está vazio. O diagrama de tempo da Figura 3.20 apresenta os dois ciclos inválidos, aonde a cache de instr. deve ignorar qualquer miss, que corresponde ao ciclo da ocorrência do flush (aonde se tem o resultado do branch/jump) indicando que o próximo endereço está incorreto, e no próximo ciclo aonde o pipeline está vazio, logo o barramento não possui um dado válido.

Figura 3.20 – Diagrama de um flush no núcleo, afetando a cache de instrução.



Fonte: Autor.

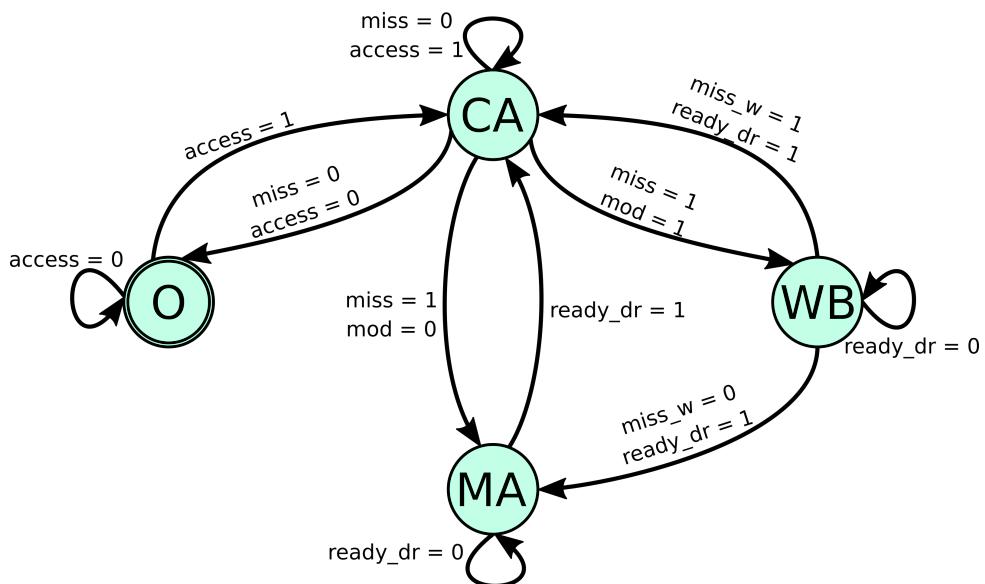
A saída da cache de instruções realiza um bypass de dados sendo escritos nela, para o registrador na saída da memória. Isso evita um ciclo a mais de latência na busca de dados da memória principal, pois mesmo quando o dado está pronto, seria necessário ainda armazena-lo na cache, e mais um ciclo ainda para acessar ele.

3.3.5 Cache de Dados

A cache de dados possuí um controle mais complexo, sendo devidamente implementado por meio de uma máquina de estados finitos, representada na Figura 3.21. Durante o estado O nenhum acesso é realizado a cache, e qualquer miss gerado pelo com-

parador de tag é ignorado, sendo trocado para o estado CA apenas quando se tem o sinal access em nível 1. Sinal esse que corresponde ao sinal de validade do endereço da cache de dados, esse é ativo apenas durante o estágio de endereçamento do núcleo (que corresponde ao EX, discutido na Seção 3.2.2) durante instr. de load e store. A condição extra para o nível ativo de access é de que o endereço também esteja dentro da região de memória que a cache de dados cobre, no caso, apenas os endereços acima de 0xFFFF_0000 são ignorados, pois correspondem a seção I/O.

Figura 3.21 – Máquina de estados finitos da cache de dados.



Fonte: Autor.

Ao reconhecer o acesso, durante o estado CA se verifica então a tag de entrada com a de saída do endereço acessado, sendo a tag de entrada obtida de um registrador temporário que armazena o endereço de acesso. Caso não ocorra um miss (tags iguais, e bit de validade em nível 1) a FSM verifica se a outro acesso ocorrendo, caso não tenha, retorna ao estado ocioso. Na ocorrência de miss é necessário verificar se o dado contido no endereço atual foi modificado, pois o mesmo deve ser salvo na memória principal para manter a coerência dos dados. Com um miss e o bit mod em nível 0, se pula para o estado MA, no qual apenas se busca um dado da memória, sem a necessidade de escrita do dado atual contido na cache. Sendo preso o estado em um loop até o término da operação de leitura da memória principal, indicado pelo sinal ready_rd.

Quando o bit de modificação está ativo, e ocorre um miss, se pula para o estado WB, aonde se realiza a escrita do dado a ser removido da cache na memória principal. Após o término da operação de escrita é pulado para o estado MA, buscando então o dado desejado (o que gerou o miss). O único caso em que o estado WB retorna para o CA é quando o miss (com mod = 1) foi resultado de uma operação SW, pois como o store word irá substituir o dado buscado da memória durante o estágio MA, se torna desnecessário

perder tempo executando tal tarefa. Importante notar que para SH e SB (half-word e byte) se tem a necessidade de pular do estado WB para o MA.

A saída da cache de dados possuí um comparador para o endereço sendo acessado atualmente, e o último acessado, para o caso de ocorrer uma sequência STORE-LOAD, por exemplo. Assim, esse comparador quando positivo (endereços iguais) realizar uma transferência do dado sendo escrito para a saída da memória, evitando um stall de um ciclo, pois o load estaria lendo o dado antigo do endereço sem esse bypass. Essa mesma lógica é utilizada quando se busca um dado da memória principal para a cache.

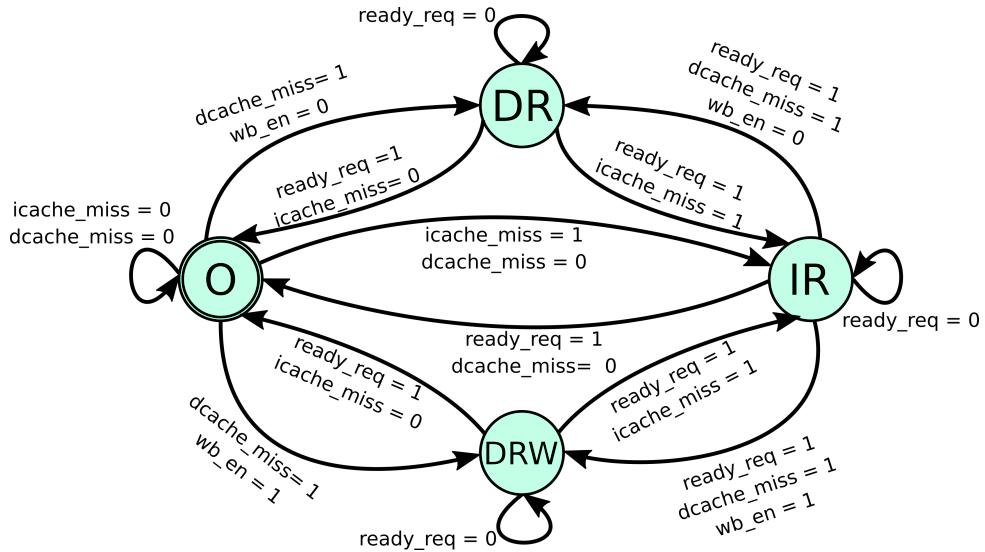
3.3.6 Controle da Cache

O controle da cache descrito aqui se refere ao seu acesso a memória principal, buscando instr. e dados, ou então realizando escritas por meio da lógica de write-back. A principal função do controle é definir qual cache irá assumir o controle sobre o acesso a memória principal, sendo no caso a prioridade estabelecida para a de dados, pois seu acesso é mais adiante no pipeline. Quando ocorre um miss na cache de instr. e na de dados simultaneamente, será acessado primeiro o endereço da de dados, armazenando em um FF o pedido de acesso da cache de instr., para posterior processamento. Esse mesmo FF também armazena pedidos da cache de dados, para a ocasião de um miss na cache de dados ocorrer enquanto se tem uma busca de instrução em andamento. Na Figura 3.22 se demonstra o funcionamento da máquina de estados finitos do controle, aonde os sinais dcache_miss e icache_miss correspondem a pedidos por dados da cache de dados, e instrução, respectivamente.

Os estados DR e DRW correspondem a pedidos de dados para a cache de dados, e pedido de escrita de dado, enquanto IR pedido de instrução. Não existe transição de um pedido da cache para ela mesma quando pronta, pois não é possível essa ocorrência no pipeline. Já transições de pedidos para a cache de instruções para a de dados, ou o contrário, existem. O sinal wb_en é utilizado para definir quando se deseja uma operação de write-back. A flag ready_req indica quando o dado da memória principal está pronto, sendo originado do controlador de memória.

Como a memória utilizada possuí operações de burst (modo síncrono) e page (modo assíncrono, leitura apenas) o controle deve enviar um sinal de cancelamento da operação atual, sempre que se troca de atendimento de pedido da cache de instr. para de dados, ou o contrário. Isso é necessário para evitar que o controle da cache receba um sinal de dado pronto do controlador da memória, que corresponde na verdade a um endereço diferente do desejado, pois os acessos burst/page são sequenciais. Sendo assim, qualquer ocasião em que se perca o fluxo sequencial de busca de dados da memória principal (que geralmente ocorre na busca de instruções) acaba por gerar o sinal de can-

Figura 3.22 – Máquina de estados finitos responsável pelo controle de acesso da cache a memória principal.



Fonte: Autor.

celamento da busca atual na memória. Outro caso além da troca de atendimento entre as cache seria o resultado de um flush no pipeline do núcleo (por desvio condicional/incondicional, exceção ou interrupção). Nessa implementação o caso a se considerar é o de leitura page, pois o controlador de memória descrito na Seção 3.4 suporta apenas esse modo, sendo o burst deixado para futuras implementações.

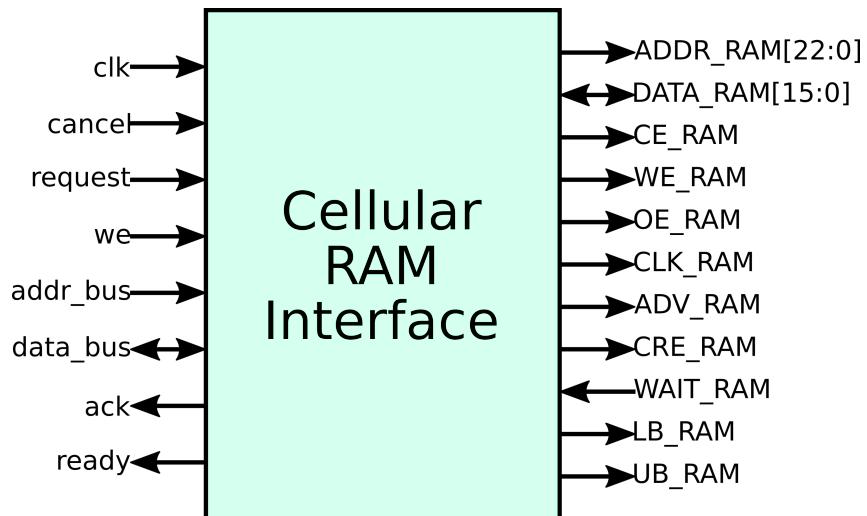
3.4 CONTROLADOR PARA CELLULAR RAM

Com os três modos de operações possíveis para a memória RAM disponível na placa Nexys3, discutida na Seção 2.4.2, projetou-se um controlador para dois modos. Ambos os modos são assíncrono, optando-se por não utilizar o síncrono nesta etapa do projeto devido a complexidade de uso do modo burst, tanto por parte do controlador abordado nesta seção, como por parte do processador em utilizar o modo burst. Para os dois modos de operação, se utiliza uma única descrição em VHDL, sendo definido no momento de síntese qual será utilizado.

A Figura 3.23 demonstra uma visão das portas da interface com a memória, aonde se tem duas entrada de controle, "stop_req" e "request". O sinal "stop_req" é responsável por cancelar uma operação atual do controlador (se tiver), e o "request" por iniciar um novo acesso com o endereço/dado atual nas portas "addr_bus" e "data_bus". O pedido de acesso é definido entre leitura/escrita de acordo com "we", que em nível '1' indica uma operação de escrita. Quando reconhecido o pedido de acesso a memória, o controlador

coloca em nível '1' por um ciclo de relógio o sinal "ack". É importante ressaltar que a interface não cancela operações durante escritas, a fim de se evitar a corrupção de dados. Quando um acesso é finalizado, a saída "ready" é colocada em nível '1', estando então a interface pronta para receber outro pedido a partir do próximo ciclo, no qual "ready" irá voltar para 0.

Figura 3.23 – Portas de entrada e saída da interface com a memória.



Fonte: Autor.

Ambas as portas de endereço e dados conectadas ao barramento são de 32 bits, sendo assim, a interface realiza dois acesso a memória a cada pedido ("request" em nível '1') realizado, pois a RAM é composta por palavras de 16 bits. Não se utiliza a lógica de habilitação do byte superior ou inferior da memória, o acesso sempre é realizado sobre uma palavra completa. As conexões com a memória RAM envolvem todos as portas necessárias para seu controle, incluindo "wait_RAM", que não fornece nenhuma informação útil em ambos os modos assíncronos, sendo apenas ignorada. Os sinais "LB_RAM" e "UB_RAM" são responsáveis pela habilitação dos bytes, que durante acessos estão sempre ativos, pois como dito anteriormente, o acesso se da sempre sobre palavras completas.

O tempo de acesso para leitura e escrita em ambos os modos está resumido na Tabela 3.6, para palavras de 32 bits. No modo de operação page demonstra-se o tempo de leitura ao acessar o final da página, e também o tempo de acesso para uma página completa para comparar com o modo padrão assíncrono. No caso, a página completa corresponde a leitura realizada 8x. As seções a seguir elaboram o acesso de ambos os modos, deixando evidente como ocorre o caso de fim de página, por exemplo. A escrita se comporta da mesma forma para os dois modos.

Tabela 3.6 – Tempo de acesso da Cellular RAM por meio do controlador projetado.

Modo	Operação	Tempo (ns)
assíncrono	leitura	210
	leitura (8x)	1680
	escrita	190
page	leitura	140
	leitura (8x)	570
	leitura (fim de pág.)	220
	escrita	190

Fonte: Autor.

3.4.1 Máquina de Estados Finitos

O CI de memória é configurado durante o tempo de energização, o qual corresponde a cerca de 150 us, no modo padrão assíncrono. Um contador interno na interface é responsável por habilitar o hardware apenas após um tempo definido de 170 us. O diagrama de estados da Figura 3.24 contém a máquina de estados utilizada para controlar a memória, sendo que no modo assíncrono padrão os estados CR1, CR2, CR3 e P nunca são utilizados, pois pertencem ao modo page. No uso da interface com o modo page, os estados CR são utilizados para realizar uma escrita nos registradores de controle da memória, configurando a mesma para esse tipo de operação (page).

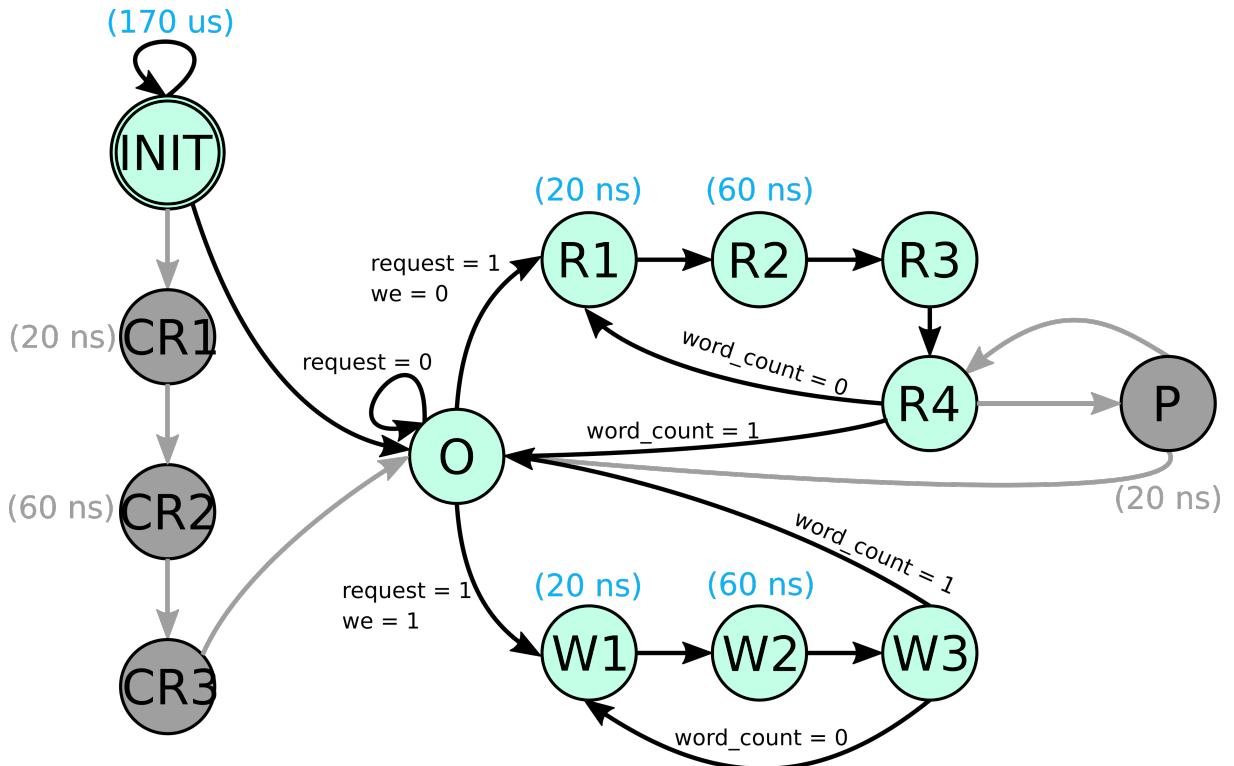
A interface foi projetada para operar na frequência de 100 MHz, logo nos estados em que não se especificou a latência, seu valor é de 10 ns. A Tabela 3.7 tem os valores de saída da interface durante cada estado de operação da FSM. Os sinais de clock e endereço válido ("clk_ram" e "adv_ram") são fixados em nível '0' durante todo o acesso assíncrono, e são ignorados.

Tabela 3.7 – Entradas e saídas da máquina de estado.

	INIT	CR1	CR2	CR3	O	R1	R2	R3	R4	P	W1	W2	W3
CE	1	0	0	1	1	0	0	0	0	0	0	0	1
WE	1	1	0	1	1	1	1	1	1	1	1	0	1
OE	1	1	1	1	1	1	0	0	0	0	1	1	1
CRE	0	1	1	0	0	0	0	0	0	0	0	0	0
LB	1	1	1	1	1	0	0	0	0	0	0	0	1
UB	1	1	1	1	1	0	0	0	0	0	0	0	1

Fonte: Autor.

Figura 3.24 – Máquina de estados finitos da interface com a memória RAM no modo assíncrono padrão, aonde os estados CR1, CR2, CR3 e P são ignorados.



Fonte: Autor.

3.4.2 Modo Assíncrono

No modo assíncrono padrão, após o término da inicialização, a FSM pula para o estado O, pois os estados de CR1 até CR3 são utilizados para configurar a memória no modo page. Durante o estado O, a mesma fica ociosa a espera de um pedido de acesso, quando se tem então a requisição de um acesso ("request" = 1), o sinal "we" define se a sequência de estados a serem executados serão a partir de R1 (para leitura) ou W1 (para escrita).

3.4.2.1 Escrita

Ao receber um pedido de escrita, no término do estado O, são armazenados em registradores temporários os dados da porta de endereço, e dados. Sendo no estado W1 iniciado o acesso a memória habilitando-se o CI por 20 ns, por fim no estado W2 realizando a escrita, ao longo de 60 ns. Quando no estado W3, que significa que a escrita foi concluída e os sinais de controle da RAM foram desabilitados, é verificado se a contagem de palavras ("word_count") tem valor '1', indicando uma palavra completa de 32 bits. Caso sim, a FSM

volta ao estado O, do contrário, se incrementa o registrador temporário do endereço em 1 para apontar ao próximo endereço.

Durante a primeira etapa da escrita se armazena os bits [15:0] do registrador de dados temporário, logo, também se multiplexa os bits de [31:16] ao incrementar o registrador de endereço, a fim de se completar a palavra. Quando se termina uma palavra, ainda no estado W3, é colocado em nível '1' o sinal "ready", para que a interface comunique ao dispositivo utilizando-a sobre o estado do acesso, que no caso está pronto. Operações de escrita não podem ser canceladas, sendo assim, durante os estados W o sinal de entrada "stop_req" é ignorado pelo controle. O tempo total de escrita de uma palavra de 32 bits, incluindo a recepção do controle (pedido de acesso), e comunicação ao dispositivo conectado sobre o término da operação, é de 190 ns.

3.4.2.2 Leitura

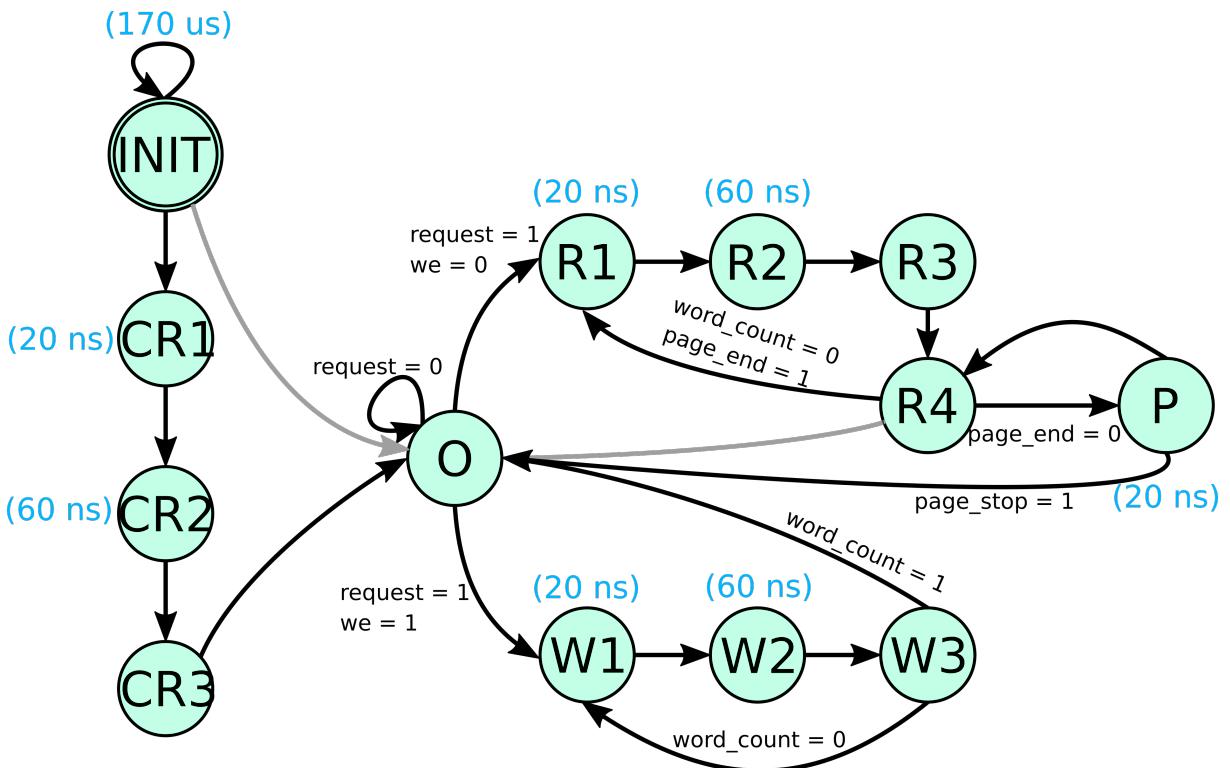
A operação de leitura utiliza apenas o registrador de endereços temporário, que possui o endereço obtido no estado O, ao receber o pedido de acesso do dispositivo. Durante o estado R1 é iniciado o acesso pela habilitação do CI, e no R2 é efetivamente lida a memória. Sendo no R3 então salvo o dado de 16 bits do barramento em um registrador temporário específico para dados de leitura. Ao chegar no estado R4, verifica-se se já foi completada uma palavra de 32 bits, por meio do sinal "word_count".

Caso "word_count" esteja ativo durante o estado R4, o sinal "ready" é colocado em nível '1' durante este mesmo ciclo para que o dispositivo que fez a requisição salve o dado disponível no barramento. Do contrário, a máquina de estados volta ao estado R1 para realizar o acesso de mais 16 bits, a fim de completar a palavra. Durante a volta para o estado R1 se incrementa em 1 o endereço temporário, enquanto que ao chegar no estado R3 novamente, se armazena agora os 2 byte lidos na parte superior do registrador de dados para leitura. A operação de leitura é cancelada por meio do sinal "stop_req" em nível '1', que irá fazer com que a FSM volte a partir de qualquer estado R, para o O. Caso o momento em que se cancelar a operação atual da interface coincida com o término de um acesso, cabe ao dispositivo que utiliza a interface decidir entre aproveitar de qualquer forma o dado disponível no barramento. O tempo total de acesso para leitura de uma palavra completa de 32 bits, desde o reconhecimento do pedido de acesso, até a disponibilização do dado no barramento, é de 210 ns.

3.4.3 Modo Page

A máquina de estados durante o modo page funciona de acordo com a Figura 3.25, aonde durante os estados CR1, CR2 e CR3 se configura o registrador de controle do CI de memória para entrar no modo page. O modo page possui o acesso para escrita da mesma forma que no modo assíncrono padrão, descrito na Seção 3.4.2.1, sendo a diferença na leitura de dados.

Figura 3.25 – Máquina de estados finitos da interface com a memória para o modo page.



Fonte: Autor.

No modo page, a memória se divide em páginas de 16 palavras de 16 bits, no qual após se realizar a leitura de uma palavra dentro da página, as demais tem um tempo reduzido de 20 ns para serem acessadas. Considerando que as palavras do dispositivo conectado são de 32 bits, é possível utilizar a interface então para buscar rapidamente 8 dados. Ao receber um pedido de acesso de leitura, é armazenado então o endereço de leitura no registrador temporário, utilizado na leitura dos 16 bits da memória durante os estados de R1 a R3, da mesma forma que no modo síncrono padrão explicado na Seção 3.4.2.2. Durante o estado R4 então se decide entre terminar o acesso a memória, ou continuar obtendo uma palavra de 16 bits a cada 20 ns no estado P.

Caso se alcance o fim da página, que é identificado quando os 4 LSB do registrador de endereço estão em nível '1', se finaliza a leitura, retornando a FSM ao estado O. Porém, no caso do término da página ocorrer no meio de uma palavra ("word_count" = 0), então se

retorna ao estado R1 para iniciar um último acesso a 16 bits, a fim de completar o dado. O acesso para apenas mais uma leitura é indicado pelo sinal "last_word", que é colocado em nível '1' nessa situação de fim de página ("page_end" = 1) com uma palavra incompleta, sendo colocado em '0' após o estado R4. Durante o estado R4, a cada palavra completada durante o acesso page, o sinal "ready" é colocado em nível '1', para indicar ao dispositivo conectado que o mesmo deve ler o dado do barramento.

Durante o estado R4 o endereço de acesso é sempre incrementado em 1, dessa forma o acesso da página é sequencial, sendo responsabilidade do dispositivo utilizando a interface de cancelar a operação da mesma por meio do sinal "stop_req" caso o endereço não coincida com o desejado da próxima leitura, ou então, caso se deseje trocar a operação para uma escrita. Durante o acesso no estado P e R4, o sinal "request" também é ignorado, logo o "ack" não é utilizado nessa situação. Assim o dispositivo conectado a interface deve estar ciente que mesmo no caso do endereço dos próximos pedidos coincidirem com o da página, ele não será reconhecido. A Figura 3.26 demonstra o acesso a uma página de 16 palavras de 16 bits, partindo de diferentes endereços inciais, resultando ou não no fim de página, e demonstrando que a mesma não reinicia o acesso após o fim da última palavra.

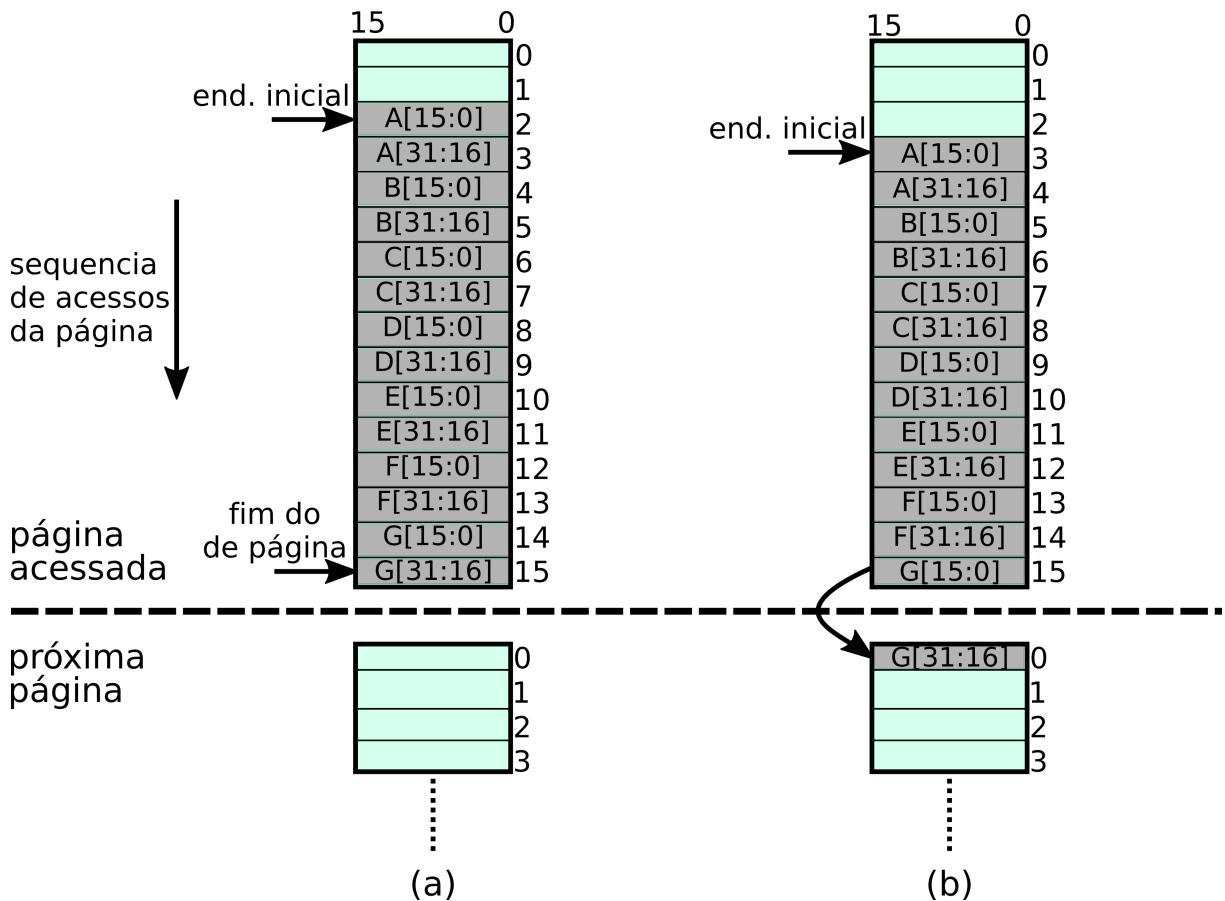
No modo page, o sinal "ready" é ativo por um ciclo durante o estado P, e não R4 como no modo assíncrono padrão. Sendo "ready" colocado em '1' a cada vez que "word_count" está ativo. A leitura de uma única palavra de 32 bits leva 140 ns, porém no caso de fim de página com palavra incompleta, devido ao retorno ao estado R1, obtém-se um tempo de 220 ns, 10 ns a mais que no modo assíncrono padrão. Isso deve ao fato de que a FSM deve pular ao estado P para indicar a conclusão da operação, em contraste com o modo padrão que o faz ainda no estado R4. O fim de página só irá ocorrer em acessos desalinhados para dados de 32 bits, logo o ideal é utilizar a memória com endereços alinhados em 4 bytes para a evitar a perda de desempenho do modo page.

O aproveitamento máximo do modo page para leitura seria no caso do acesso se dar no primeiro endereço da página (LSB igual a '0'), em que se teria um total de 570 ns para 8 palavras de 32 bits. No modo assíncrono padrão, o tempo de leitura da mesma quantidade de endereços levaria 1680 ns.

3.5 DISPOSITIVOS I/O

Os dispositivos I/O utilizados continuam os mesmos, sendo uma UART para realizar comunicações serial com o computador pessoal, e um temporizador de 32 bits. Considerou-se adicionar funções extras ao temporizador, porém o RISC-V possuí instruções de controle dedicadas ao acesso de contadores de ciclos de relógio, entre outras finalidades, sendo assim procurou-se não adicionar nenhuma complexidade ao mesmo,

Figura 3.26 – Demonstração de acesso da página. A Figura (a) contém um acesso alinhado, que resulta no fim da página com palavra completa, e consequentemente não é necessária a leitura de uma palavra adicional da próxima página. Já no exemplo (b), ao terminar a página se tem uma palavra incompleta, se tornando necessário um acesso extra apenas para completa-lá.



Fonte: Autor.

que se tem como ideal ser substituído futuramente. Entre os motivos de não ter se adicionado justamente os temporizadores descrito pelo RISC-V, destaca-se o planejamento da inserção deles no processador, pois além de serem acessados por meio de instruções para registradores de controle/status (CSR), eles também são de 64 bits, independente da arquitetura (RV32I, RV64I ou RV128I).

O endereço inicial dos dispositivos mapeados diretamente na memória é o 0xFFFF_0000, apresentado na Seção 3.1.3, ressaltando as modificações nos endereços da UART e do Timer dentro da seção. Anteriormente se tinha uma atribuição em que cada registrador acessado da UART (controle ou dado) estava alinhado por 4 bytes, conforme o tamanho da palavra do sistema. Por mais que o registrador de controle da mesma tivesse apenas 4 bits úteis, e o dado transmitido/recebido pelo dispositivo no seu registrador de dados fosse de 8 bits.

Era essencial o alinhamento de 4 bytes pois o processador utiliza acessos alinhados apenas, e além do núcleo pipeline, se tinha um multicíclico que usava dos mesmos dispositivos I/O e que não possuía instruções de LH/LB (load half-word / load byte), nem de escrita. Porém agora o foco é o núcleo pipeline que implementa tais instruções, sendo assim o novo mapeamento dos dispositivos ficou conforme a Tabela 3.8, em que se percebe cada registrador da UART de 1 byte agora ocupando apenas um endereço, assim como o seu novo registrador de configuração de baudrate. O temporizador como um registrador de 32 bits é alinhado ao próximo endereço de 4 bytes.

Tabela 3.8 – Endereços de acesso dos dispositivos I/O internos do processador.

Dispositivo	Endereço
UART - Regs. de Controle/Status	0xFFFF_0000
UART - Regs. de Dados	0xFFFF_0001
UART - Regs. de Baudrate	0xFFFF_0002
Temporizador 0	0xFFFF_0004

Fonte: Autor.

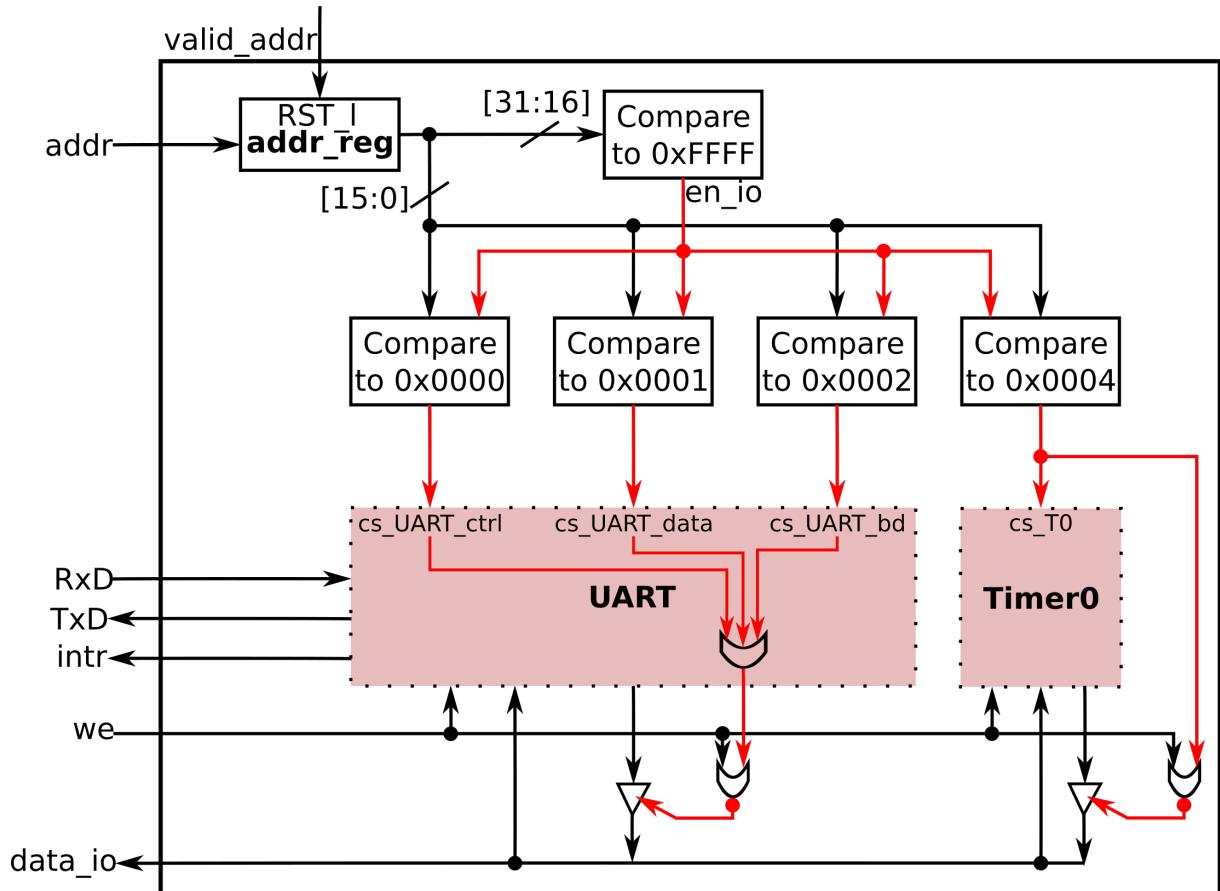
3.5.1 Controle

O controle do acesso aos dispositivos I/O continua com o mesmo funcionamento, exceto pelo fato de que agora o sinal de habilitação da seção I/O foi removido do decodificador principal da memória, e se encontra junto aos demais comparadores do controle I/O. O diagrama de blocos de Figura 3.27 demonstra a decodificação do acesso, aonde se tem a necessidade de registrar o endereço para uso no próximo ciclo. Isso se deve ao fato do endereçamento da memória de dados ter sido adiantado um ciclo no núcleo, para compensar a latência de 1 ciclo de leitura dos blocos de RAM, conforme explicado na Seção 3.2.2. Porém os dispositivos I/O (UART e Timer0) possuem a leitura assíncrona, se tornando necessário então o armazenamento do endereço alvo por um ciclo.

Para evitar acessos indevidos, o sinal valid_addr reseta o registrador do endereço quando não se deseja realizar uma operação de load/store. Supondo um acesso válido (valid_addr = 1), temos então no próximo ciclo a comparação dos 16 MSb do endereço com o correspondente atribuído a seção I/O, tendo como saída o sinal en_io ativo em 1. Se o en_io resultar em 0, o restante dos comparadores são ignorados, pois não é um endereço localizado dentro da seção I/O, do contrário, se define qual é o dispositivo a ser lido/escrito de acordo com os 16 LSb. Internamente a UART multiplexa sua entrada/saída para o barramento de dados do bloco I/O, de acordo com o sinal de seleção, notando que todos os sinais de seleção após o primeiro comparador, são ativos em 0.

Após a leitura da UART/Timer, verifica-se se não é uma operação de escrita pelo

Figura 3.27 – Controle interno do acesso aos dispositivos I/O.



Fonte: Autor.

sinal we, caso seja, o buffer tri-state desabilita a conexão da porta de saída com o barramento, para evitar conflitos de dados. A única modificação real no controle (além de se ter movido o sinal en_io do decodificador da memória) foi a adição do pino de interrupção. Quando se tem em nível 1 a saída intr, significa que uma interrupção da seção I/O foi solicitada, que no caso pertence necessariamente a USART, já que é o único dos dois dispositivos com essa função. A USART realiza internamente a lógica de geração do sinal de interrupção, sendo por responsabilidade do controle da seção I/O administrar todas as fontes de interrupção. Novamente, se tem apenas uma fonte de interrupção (a USART), então desconsiderou-se nessa etapa do projeto a implementação de um controlador de interrupções.

3.5.2 UART

O registrador de controle da UART continua utilizando apenas 5 bits, sendo deixados os demais conectados diretamente ao GND, aberto a futuras modificações. A funcionalidade de cada bit é apresentada na Figura 3.28, destacando-se que por parte do programador, deve-se utilizar os bits [0] e [1] para saber quando é possível escrever e ler dados, respectivamente, no dispositivo, caso não se utilize interrupções. Os campos RX_R e TX_R, que indicam quando uma recepção/transmissão está finalizada, são modificados únicamente pelo hardware, e consequentemente seu acesso só é válido para leitura, escritas serão ignoradas. A técnica de polling para acesso ao dispositivo pode ser feita verificando-se pelo nível 1 nesses campos, de acordo com a função desejada (enviar/receber dado).

Figura 3.28 – Registrador de controle da UART.

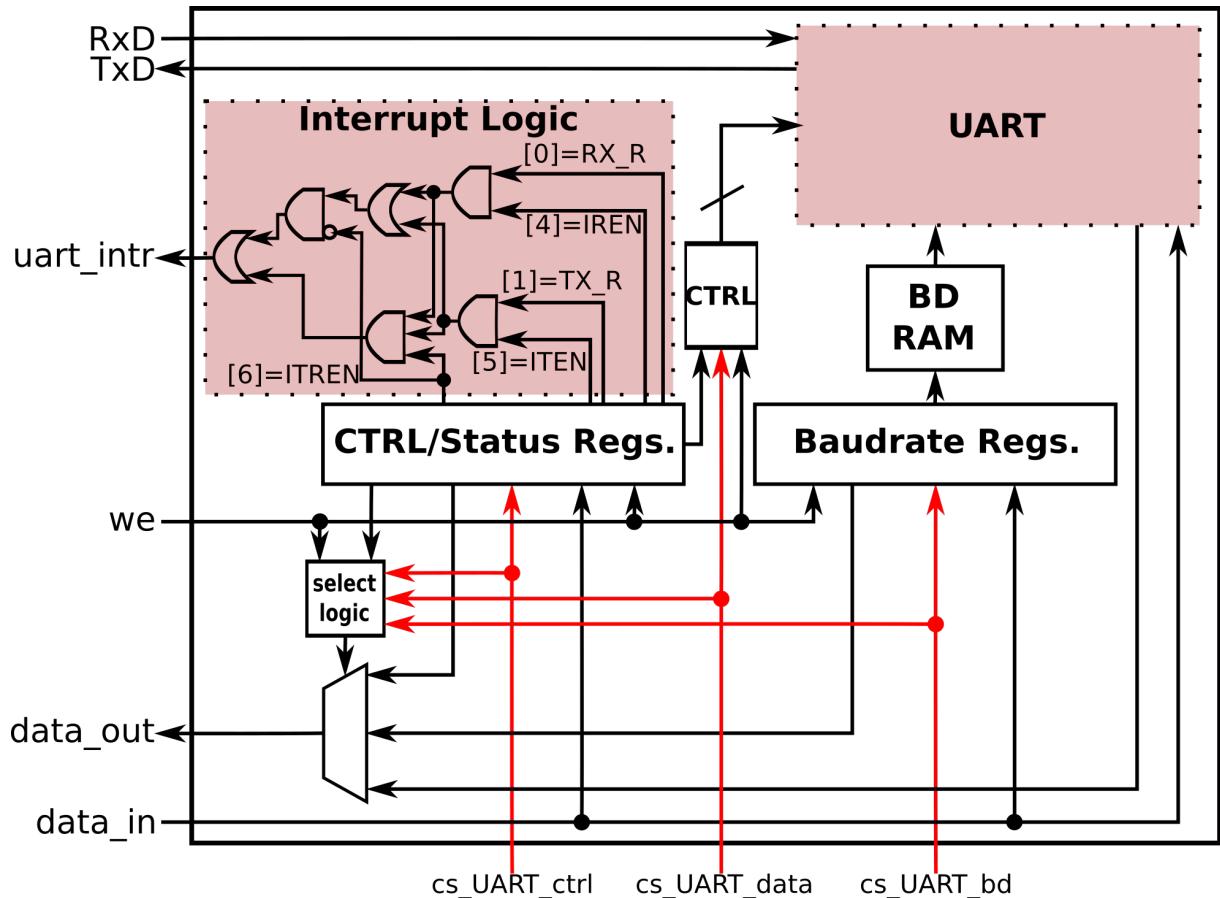
	7	6	5	4	3	2	1	0
Type	RES	ITREN	ITEN	IREN	RES	TX_R	RX_R	
Reset	RO	R/W	R/W	R/W	RO	RO	R/W	R/W
	0	0	0	0	0	0	0	0

Fonte: Autor.

Nos campos [4] e [5] adicionou-se as flags de habilitação de interrupções, separadas de acordo com a função desejada, recepção ou transmissão, respectivamente. O bit [6] habilita a interrupção de leitura/escrita simultânea, ou seja, a interrupção só ocorrerá quando ambos os dados estiverem prontos, sendo nesse caso, ignorados os campos [4] e [5] por meio de uma porta AND com o sinal ITREN negado (bit [6]). Na Figura 3.29 se apresenta a lógica de acesso aos registradores da UART, e da geração de interrupções. O hardware de recepção/transmissão ao finalizar uma de suas funções, coloca em nível 1 o bit correspondente no registrador de controle, que passa por uma porta AND, a fim de desabilitar a sinalização de dado pronto, caso a interrupção correspondente esteja desabilitada. Não se diferencia o tipo de interrupção sinalizada, sendo atribuído ao software definir qual se deseja acessar, sendo uma sugestão a atribuição de prioridade na verificação entre envio e leitura de dados.

A seleção da saída de dados durante uma leitura é feita de acordo com o sinal de seleção de entrada, discutido na Seção 3.5.1, no qual se optou por adicionar o registrador de baudrate saída padrão, caso se tente acessar o registrador de dados e o mesmo não tenha um dado válido. Em acessos ao registrador de dados para escrita, ignora-se sua operação, caso TX_R = 0, já que a última transmissão solicitada ainda está em andamento. O registrador de dados está ocultado, pois o mesmo faz parte do bloco de transmissão e recepção da UART, e não de seu controle.

Figura 3.29 – Controle da UART.



Fonte: Autor.

O controle de baudrate era feito diretamente na descrição VHDL, sendo impossível a configuração do dispositivo durante a execução de um programa, agora se alocou no endereço 0xFFFF_0002 um registrador de 1 byte, que de acordo com sua codificação, irá definir a partir do próximo ciclo de relógio (após a realização da escrita) o baudrate da UART. O registrador acessa um pequeno bloco de memória dedicada implementada em RAM distribuída, sem essencialmente uma ROM com os possíveis valores a serem configurados a UART. Seus possíveis valores são apresentados na Tabela 3.9, sendo válidos acesso tanto para leitura quanto para escrita, ressaltando que apenas os 3 LSb definem a taxa de comunicação, os demais são ignorados. O valor padrão de inicialização do baudrate da UART (durante a configuração da FPGA) é de 19200.

Tabela 3.9 – Codificação do registrador de baudrate da UART.

UART - Registrador de Baudrate	
[2:0]	Baudrate
000	1200
001	2400
010	4800
011	9600
100	19200
101	38400
110	57600
111	115200

Fonte: Autor.

4 RESULTADOS E DISCUSSÃO

4.1 SÍNTESE E IMPLEMENTAÇÃO

O uso do dispositivo FPGA utilizado no projeto, Spartan-6, pode ser conferido na Tabela 4.1, aonde se destaca os principais parâmetros. A frequência de operação obtida ficou dentro do desejado, 100 MHz, que é a disponível na placa de desenvolvimento Nexys3. O número de blocos de RAM utilizados limitou-se a 28, por ser um número que apresenta um bom aproveitamento da memória com o mapeamento utilizado para a cache (mapeamento direto). Uma aplicação para os 4 blocos restantes seria seu uso como possíveis buffers de dados para a cache (entre a cache e a memória principal), buffer para dispositivos I/O (UART), ou então modificações na forma como se implementa a BHT do núcleo, que permaneceu inalterada. O Apêndice A contém as opções de síntese utilizadas no software ISE Design 14.7, e também mais detalhes sobre os dados finais obtidos na etapa Place and Route.

Tabela 4.1 – Resumo de utilização de recursos da FPGA Spartan-6.

Recursos	Disponível	Usado	Utilização (%)
Número de Slice Registers	18,224	1,115	6%
Número de Slices LUTs	9,112	2,553	28%
Número de Slices Ocupados	2,278	894	39%
Blocos de RAM	32	28	87.5%
Frequência Máxima (MHz)		100.482	

Fonte: Autor.

O consumo de lógica em geral aumentou pouco em relação ao obtido na etapa anterior do projeto, revisada na Seção 2.2, demonstrando o impacto da melhoria na descrição VHDL. Outros fatores também incluem o uso de um barramento único para o núcleo, que agora dividiu-se em dois conectados a cache de instrução, e de dados. Essa alteração alivia a latência, pois antes cada bloco de RAM (que possui localização fixa na FPGA) era conectada diretamente a dois estágios diferentes do núcleo. Logo, a ferramenta de análise do tempo tinha que satisfazer a latência de 10 ns para dois caminhos diferentes, distantes entre si no pipeline (estágio 1 e 4), para cada BRAM utilizado como memória do sistema. Outros tópicos a respeito da análise de tempo (10 ns) são abordados nas subseções a seguir.

4.1.1 Análise I/O

A implementação do controle da cache, apesar de não ser tão simples, não foi um acréscimo de lógica muito grande em comparação com o simples decodificador que se utilizava antes. O maior desafio para o alcance dos 100 MHz estava de fato no uso do controlador da RAM. Seu barramento de saída de 16 bits bidirecional constantemente falhava na restrição de tempo ajustada. Comportamento esperado, pois uma das maiores latências obtidas na configuração FPGA é justamente nas conexões I/O, em especial com lógica tri-state. Também é importante notar que todos os pinos I/O utilizados são definidos pelas conexões da FPGA com os Cls (UART e Memória RAM) disponibilizados na placa de desenvolvimento Nexys-3, não sendo possível alterar essa questão do projeto.

A frequência de operação obtida é referente a propagação de um sinal por meio de um caminho combinacional, entre dois FF. Os parâmetros de tempo obtido para as conexões I/O são demonstradas na Tabela 4.2, que dizem respeito a propagação do pino até o FF. Caso o pino I/O se conecte a mais de um FF, demonstra-se o pior caminho possível dele.

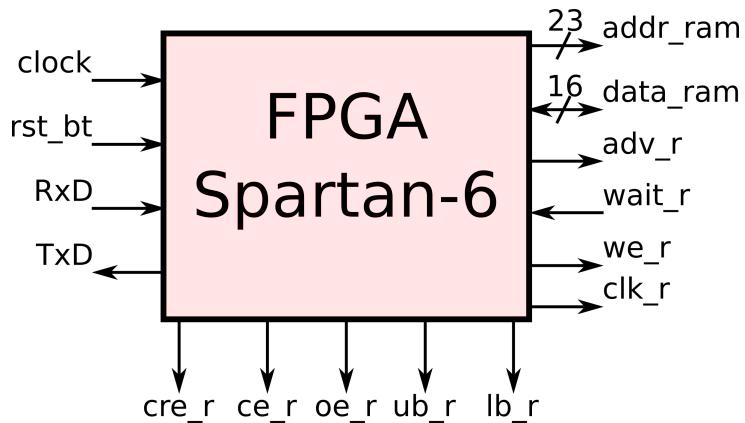
Tabela 4.2 – Offset in e offset out obtidos.

Entrada	Setup (ns)
rst_bt	0.881
data_ram[0]	4.278
Saída	Latência (ns)
addr_ram[22]	8.943
data_ram[6]	9.863
ce_r	9.479
cre_r	8.950
lb_r	9.750
oe_r	8.963
ub_r	9.617
we_r	9.139

Fonte: Autor.

Devido a baixa velocidade de transferência da UART, não foi atribuído nenhuma restrição de tempo aos pinos RxD e TxD, a fim de não sobrecarregar a ferramenta de otimização com ações desnecessárias. Para o barramento de dados e endereço da RAM demonstra-se apenas o tempo de um caminho, o pior (mais lento) obtido entre todos os 23 bits de endereço, e também dos 16 bits de dados. As conexões de controle da RAM externa, wait_r e clk_r são ignoradas, pois a memória é utilizada no modo assíncrono, assim como a saída adv_r, que é utilizada permanentemente com o nível lógico 0, conforme explicado na Seção 3.4. A Figura 4.1 contém todos os pinos de entrada e saída implementados. Todos os sinais com o prefixo '_r' são referentes a sinais de controle da RAM externa.

Figura 4.1 – Entradas e saídas implementadas na FPGA.



Fonte: Autor.

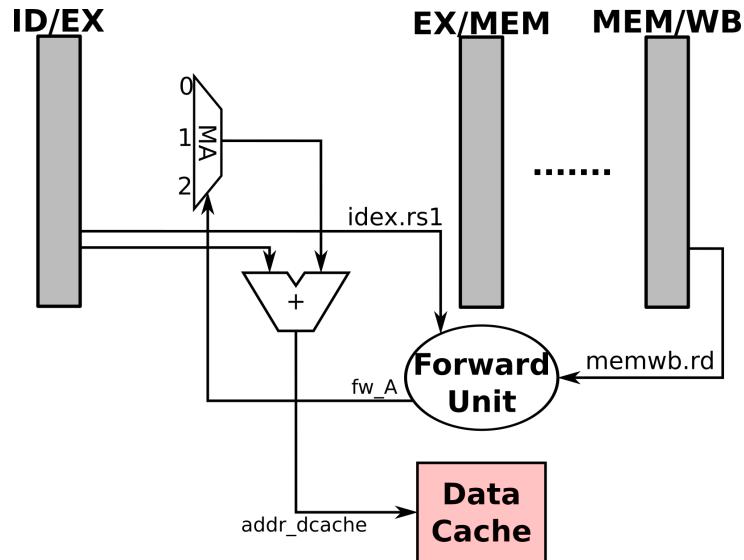
4.1.2 Análise de Caminho Crítico do Núcleo

O caminho crítico de operação do núcleo é definido pelo caminho de operação começando no registrador de pipeline ID/EX, e terminando no controle da cache de dados, totalizando 9.952 ns (100.482 MHz). A Figura 4.2 demonstra sua propagação, saíndo do regs. de pipeline no sinal idex.rs1, que é utilizado pela unidade de forward a fim de se verificar se corresponde ao destino de escrita do estágio WB (memwb.rd). Após a tomada de decisão se envia o sinal de controle fw_A para definir o operador a ser somado com o imediato, formando assim o endereço alvo de acesso da cache. Considerando apenas os FF internos do núcleo (sem a cache), a maior latência se encontra efetivamente no estágio ID/EX, saíndo o registrador ID e terminando no EX. Passando pela operação de shift na ALU, totalizando 9.755 ns.

Todos os FF do núcleo foram analisados com o objetivo de se obter um máximo de 10 ns de propagação entre eles, com exceção do banco de registradores, que no estágio ID deve fornecer o dado acessado ainda no mesmo ciclo, com leitura assíncrona. Estabeleceu-se então 4 caminhos gerais possíveis que envolvem o componente, que podem ser visualizados na Figura 4.3, nomeados de C1 a C3. O caminho C1 corresponde ao endereçamento do banco, começando no registradores IF/ID, e terminando no registrador alvo de leitura do banco. Continuando com o acesso ao banco no estágio ID, o caminho C2 é a propagação do dado lido no registrador rs1, até o registrador de pipeline ID/EX, tendo no caminho um MUX. Nesse exemplo, o regs. rs1 acessado foi transferido para a saída no MUX, para realizar o caminho. Ambos C1 e C2 foram definidos em 5 ns, obtendo-se para cada um, respectivamente 4.230 e 2.827 ns.

A propagação do caminho C3 corresponde a escrita no banco, partindo do estágio WB. Divide-se em dois pontos finais possíveis, o registrador destino do banco, e o registrador de pipeline ID/EX, pois o banco realiza a lógica de forward interna, elaborada na Seção 3.2.6. Ambos os pontos finais (a) e (b) do caminho C3 foram analisados em conjunto com

Figura 4.2 – Caminho crítico do núcleo.



Fonte: Autor.

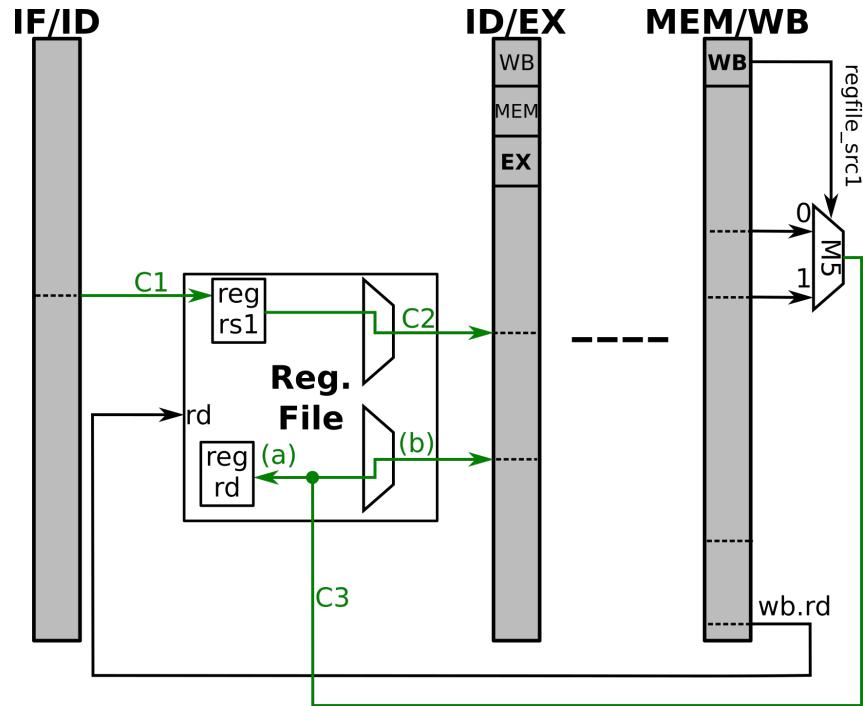
o restante do núcleo com tempo de propagação máximo em 10 ns. Vale ressaltar que a habilitação de escrita do banco de regs. está ocultada, mas ela influência no caminho C3 para controle do MUX de saída do banco, e por consequência, também no caminho C2. Considerou-se nesse exemplo que o regs. rs1 foi acessado sem forward interno, enquanto o rs2 teve transferido para a saída o dado a ser escrito no banco.

4.2 PROGRAMA TESTE

A fim de se obter um parâmetro simples de desempenho do processador, se realizou uma multiplicação entre duas matriz de dimensões iguais. O código foi escrito em C, e compilado com nível de otimização -O1, o programa completo pode ser verificado no Apêndice B. Para medir o tempo de execução se utilizou do temporizador da seção I/O, que conta o número de ciclos passados desde a ocorrência do último reset. Como o registrador temporizador é de 32 bits, operando a 100 MHz o maior intervalo de tempo que se obtém nele é de aproximadamente 43 segundos.

É importante notar que a operação de multiplicação não é suportada por hardware, visto que a extensão M (que prove essas instr.) não foi implementada. Assim, sua execução é realizada inteiramente por software. Além das multiplicações, também é comum encontrar outro tipo de operação em programas, que o processador não suporta, sendo ponto flutuante. O uso de operadores em ponto flutuante requer hardware dedicado, incluindo banco de registradores e FPU. Na execução de operações em ponto flutuante também se tem então o tratamento por software da função, sendo visível o im-

Figura 4.3 – Caminhos de propagação do banco de registradores.



Fonte: Autor.

pacto no desempenho.

Definiu-se então o tamanho de ambas as matrizes em 100x100, assim como a final obtida, totalizando 1 milhão de multiplicações. Por fim, realizou-se a rotina com o uso de dados do tipo inteiro, e ponto flutuante. Com inteiros, obteve-se um tempo de execução de 2.65439 segundos, e para ponto flutuante 4.61273935. Em outros termos, os resultados foram de aproximadamente 376.735k multiplicações por segundo com dados inteiro, e 216.791k em ponto flutuante. O trecho do programa que calcula a multiplicação foi escrito conforme:

```
// get timer value
timer0_read(t1);

//-----
// Mc = Ma * Mb
sum = 0;
if (cola != rowb){}
else
{
    for (c = 0; c < rowa; c++)
    {
        for (d = 0; d < colb; d++)

```

```
{  
    for (k = 0; k < rowb; k++)  
    {  
        sum = sum + Ma[c][k]*Mb[k][d];  
    }  
    Mc[c][d] = sum;  
    sum = 0;  
}  
}  
  
// get timer value  
timer0_read(t2);
```

5 CONCLUSÃO

Neste trabalho foram implementadas modificações sobre o processador RV32EC previamente obtido. Dentre as alterações, no núcleo limitou-se a modificar apenas o essencial para adicionar suporte as instruções de acesso a registradores de controle, que não estavam presentes, e a modificação da base E para a I. Apesar de não implementadas anteriormente (instr. CSR), não necessitou-se de nenhuma alteração grande no pipeline (como acréscimo de estágios), pois o fato é que o conjunto de instruções implementados do RISC-V permaneceu o mesmo, visto que a base E consiste nas mesmas instruções que a base I. A mudança da base teve como sequência o aumento do tamanho do banco de regs., decisão tomada considerando que a FPGA é um dispositivo rico em FFs. O uso dos CSR possibilitou a implementação da lógica para tratamento de exceções/interrupções, que se aproveitaram do caminho do flush de desvios incondicionais. Simplificando assim o controle adicionado ao núcleo para executar essas operações. Ressaltasse também a remoção da extensão C do RISC-V, que devido a sua forma de implementação pode ser facilmente adicionada/removida do hardware, podendo ser futuramente uma configuração do processador o seu uso ou não, por meio de registradores de controle extras.

A sessão dos dispositivos I/O continuou com apenas o uso da UART para troca de informações com o computador pessoal, e um temporizador, sendo adicionada lógica de interrupção apenas a UART. Deixando-se intencionalmente o temporizador sem desenvolvimento, visto que futuramente se visa sua substituição pelos contadores/temporizadores próprios da ISA do RISC-V. Um ponto importante da UART, que não foi um problema durante o projeto devido ao (baixo) tamanho dos programas escritos, é sua velocidade de transferência. Programas de até poucos MBytes poderiam levar minutos para serem transferidos do computador pessoal para a memória principal da placa Nexys-3, por meio da UART. Futuramente se deseja trocar essa forma de transferência de programas para o sistema, utilizando uma mais rápida.

O componente que mais sofreu modificações foi a memória interna do processador, que antes era representada como a memória principal do sistema. Modificando-se para uma Cache L1 de dados e instruções separadas, caracterizando uma arquitetura de harvard entre o núcleo e a cache. Seu tamanho de 16 kiB de dados e de instr. foi obtido pelo mapeamento direto, aonde considerou-se a implementação da associação por conjunto. Porém optou-se nesta etapa do projeto por uma maior capacidade de armazenamento, e um controle simplificado de implementação para a cache. Sua coerência com a memória principal foi estabelecida por meio da política de write-back, sem opção de escrita direta na memória (write-through). A falta da capacidade de se escrever direto na memória (write-through) acaba por causar a incoerência da memória de dados com a de instrução em programas que se auto-modificam em tempo de execução.

Projetou-se um controlador de memória assíncrona para se utilizar com a Cellular RAM disponível na placa Nexys-3. Dentre seus modos de operação, o utilizado é o page, que possibilita leituras sequenciais rápidas na memória em endereços adjacentes. A escrita por sua parte, é realizada uma por vez, não havendo ganho em relação ao modo assíncrono padrão. Outro modo de operação disponível era o burst, que apesar de possuir uma leitura com mesmo desempenho que o page, possibilitava escritas em sequência na memória, com tempo de acesso reduzido. Com o controlador capaz de operar a memória no modo assíncrono padrão e page, o próximo passo é a adição do controle síncrono (burst) da memória. Não se implementou nenhum tipo de buffer na memória cache, que geralmente se encontra na operação de escrita. Se deixou esse hardware para ser descrito em conjunto com o modo de operação burst da memória RAM. Visto que o modo page utilizado não possibilita escritas sequências rápidas na memória, ganhando tempo apenas na leitura.

O programa teste executado verificou a capacidade de execução do processador em operações de multiplicação, envolvendo dados do tipo inteiro e ponto flutuante. Porém é importante notar que o resultado é um parâmetro que deve ser aprofundado, pois diversos outros fatores do sistema não foram quantificados, como a taxa de miss e hit da memória cache. Entre as razões da não realização de testes mais profundos está a ausência de registradores de desempenho do processador, pois é comum os processadores terem contadores dedicados a número de instruções executadas, wall-clock, entre outros. Assim, é essencial que se implemente futuramente tais registradores, para que seja possível realizar testes mais precisos a respeito da performance do hardware.

5.0.1 Sugestões para Trabalhos Futuros

5.0.1.1 Núcleo

- Adição de outras extensões do RISC-V, como a M de multiplicação, implicando em possíveis modificações na estrutura do pipeline;
- Implementação de mais CSR (regs. de controle/status) de nível M, como o mie (interrupções pendentes), e/ou contadores de desempenho do processador.

5.0.1.2 Memória Cache

- Implementação de coerência entre a cache de dados e instrução;
- Configuração por hardware do uso da escrita na memória principal por write-back ou write-through;
- Adição de buffers tanto de leitura quanto escrita, para se utilizar com memórias que disponibilizam modos de operação page/burst.

5.0.1.3 Seção I/O

- Modificação do temporizador atualmente implementado, nos registradores contadores do RISC-V mapeados diretamente na memória;
- Uso de interrupções em outros dispositivos além da UART, como os próprios temporizadores implementados;
- Estudo de outras formas para se transferir programas para o sistema, a fim de compensar a baixa velocidade da UART.

REFERÊNCIAS BIBLIOGRÁFICAS

ARCHANA, M.; KASHWAN, K. R. An efficient architecture for improved reliability of cache memory using same tag bits. In: **2016 International Conference on Recent Trends in Information Technology (ICRTIT)**. IEEE, 2016. p. 1–4. ISBN 978-1-4673-9802-2. Disponível em: <<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=7569568>>;<<https://ieeexplore.ieee.org/document/7569568>>.

ARORA, H.; KOTECHA, S.; SAMYAL, R. Dynamic branch prediction modeller for risc architecture. In: **2013 International Conference on Machine Intelligence and Research Advancement**. [S.I.]: IEEE, 2013. p. 397–401. ISBN 978-0-7695-5013-8.

DIGILENT. **Nexys3 Board Reference Manual**. [S.I.], 2013. 22 p. Disponível em: <https://reference.digilentinc.com/_media/reference/programmable-logic/nexys-3/nexys3_rm.pdf>.

KIM, Y. kuen; SONG, Y. H. Impact of processor cache memory on storage performance. In: **2017 International SoC Design Conference (ISOCC)**. IEEE, 2017. p. 304–305. ISBN 978-1-5386-2285-8. Disponível em: <<https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=8368908>>;<<https://ieeexplore.ieee.org/document/8368908>>.

MICRON. **Async/Page/Burst CellularRAM 1.5: MT45W8MW16BGX**. [S.I.]. 68 p.

MORAIS, K. P. **Descrição em VHDL de um Processador RISC-V RV32EC**. 2018. 403 p. Disponível em: <<https://repositorio.ufsm.br/handle/1/15410>>.

PATTERSON, D. A.; HENNESSY, J. L. **Computer Organization and Design - The Hardware Software/Interface**. 3. ed. [S.I.]: Morgan Kaufmann, 2004. 656 p. (The Morgan Kaufmann Series in Computer Architecture and Design). ISBN 1558606041.

WATERMAN, A.; ASANOVIĆ, K. **The RISC-V Instruction Set Manual, Volume I: User-Level ISA, Document Version 2.2**. [S.I.], 2017. 145 p. Disponível em: <<https://github.com/riscv/riscv-isa-manual/blob/master/release/riscv-spec-v2.2.pdf>>.

_____. **The RISC-V Instruction Set Manual, Volume II: Privileged Architecture, Document Version 1.10**. [S.I.], 2017. 91 p. Disponível em: <<https://github.com/riscv/riscv-isa-manual/blob/master/release/riscv-privileged-v1.10.pdf>>.

XILINX. **Xilinx XST User Guide**. [S.I.], 2009. 485 p. Disponível em: <https://www.xilinx.com/support/documentation/sw_manuals/xilinx11/xst.pdf>.

_____. **Spartan-6 FPGA Block RAM Resources - User Guide**. [S.I.], 2011. 34 p. Disponível em: <https://www.xilinx.com/support/documentation/user_guides/ug383.pdf>.

APÊNDICE A – DADOS DE SÍNTSESE E IMPLEMENTAÇÃO

A.1 – SYNTHESIS REPORT: SYNTHESIS OPTIONS SUMMARY

---- Source Parameters

Input File Name : "top_module.prj"
Ignore Synthesis Constraint File : NO

---- Target Parameters

Output File Name : "top_module"
Output Format : NGC
Target Device : xc6slx16-3-csg324

---- Source Options

Top Module Name : top_module
Automatic FSM Extraction : YES
FSM Encoding Algorithm : Auto
Safe Implementation : No
FSM Style : LUT
RAM Extraction : Yes
RAM Style : Auto
ROM Extraction : Yes
Shift Register Extraction : YES
ROM Style : Auto
Resource Sharing : YES
Asynchronous To Synchronous : NO
Shift Register Minimum Size : 2
Use DSP Block : Auto
Automatic Register Balancing : No

---- Target Options

LUT Combining : Auto
Reduce Control Sets : Auto
Add IO Buffers : YES
Global Maximum Fanout : 100000
Add Generic Clock Buffer(BUFG) : 16
Register Duplication : YES

Optimize Instantiated Primitives : NO
 Use Clock Enable : Auto
 Use Synchronous Set : Auto
 Use Synchronous Reset : Auto
 Pack IO Registers into IOBs : Auto
 Equivalent register Removal : YES

---- General Options

Optimization Goal : Speed
 Optimization Effort : 1
 Power Reduction : NO
 Keep Hierarchy : No
 Netlist Hierarchy : As_Optimized
 RTL Output : Yes
 Global Optimization : AllClockNets
 Read Cores : YES
 Write Timing Constraints : NO
 Cross Clock Analysis : NO
 Hierarchy Separator : /
 Bus Delimiter : <>
 Case Specifier : Maintain
 Slice Utilization Ratio : 100
 BRAM Utilization Ratio : 100
 DSP48 Utilization Ratio : 100
 Auto BRAM Packing : NO
 Slice Utilization Ratio Delta : 5

A.2 – PLACE AND ROUTE REPORT: DEVICE UTILIZATION SUMMARY

Device Utilization Summary:

Slice Logic Utilization:

Number of Slice Registers:	1,115	out of 18,224	6%
Number used as Flip Flops:	1,115		
Number used as Latches:	0		
Number used as Latch-thrus:	0		
Number used as AND/OR logics:	0		

Number of Slice LUTs:	2,553 out of 9,112	28%
Number used as logic:	2,484 out of 9,112	27%
Number using 06 output only:	2,103	
Number using 05 output only:	154	
Number using 05 and 06:	227	
Number used as ROM:	0	
Number used as Memory:	60 out of 2,176	2%
Number used as Dual Port RAM:	60	
Number using 06 output only:	16	
Number using 05 output only:	0	
Number using 05 and 06:	44	
Number used as Single Port RAM:	0	
Number used as Shift Register:	0	
Number used exclusively as route-thrus:	9	
Number with same-slice register load:	1	
Number with same-slice carry load:	8	
Number with other load:	0	

Slice Logic Distribution:

Number of occupied Slices:	894 out of 2,278	39%
Number of MUXCYs used:	384 out of 4,556	8%
Number of LUT Flip Flop pairs used:	2,613	
Number with an unused Flip Flop:	1,550 out of 2,613	59%
Number with an unused LUT:	60 out of 2,613	2%
Number of fully used LUT-FF pairs:	1,003 out of 2,613	38%
Number of slice register sites lost to control set restrictions:	0 out of 18,224	0%

A LUT Flip Flop pair for this architecture represents one LUT paired with one Flip Flop within a slice. A control set is a unique combination of clock, reset, set, and enable signals for a registered element.

The Slice Logic Distribution report is not meaningful if the design is over-mapped for a non-slice resource or if Placement fails.

IO Utilization:

Number of bonded IOBs:	52 out of 232	22%
Number of LOCed IOBs:	52 out of 52	100%

Specific Feature Utilization:

Number of RAMB16BWERS:	28 out of	32	87%
Number of RAMB8BWERS:	0 out of	64	0%
Number of BUFI02/BUFI02_2CLKs:	0 out of	32	0%
Number of BUFI02FB/BUFI02FB_2CLKs:	0 out of	32	0%
Number of BUFG/BUFGMUXs:	1 out of	16	6%
Number used as BUFGs:	1		
Number used as BUFGMUX:	0		
Number of DCM/DCM_CLKGENs:	0 out of	4	0%
Number of ILOGIC2/ISERDES2s:	0 out of	248	0%
Number of IODELAY2/IODRP2/IODRP2_MCBs:	0 out of	248	0%
Number of OLOGIC2/OSERDES2s:	0 out of	248	0%
Number of BSCANs:	0 out of	4	0%
Number of BUFHs:	0 out of	128	0%
Number of BUFPLLs:	0 out of	8	0%
Number of BUFPLL_MCBs:	0 out of	4	0%
Number of DSP48A1s:	0 out of	32	0%
Number of ICAPs:	0 out of	1	0%
Number of MCBs:	0 out of	2	0%
Number of PCILOGICSEs:	0 out of	2	0%
Number of PLL_ADVs:	0 out of	2	0%
Number of PMVs:	0 out of	1	0%
Number of STARTUPs:	0 out of	1	0%
Number of SUSPEND_SYNCs:	0 out of	1	0%

Overall effort level (-ol): High

Router effort level (-rl): High

APÊNDICE B – PROGRAMA TESTE

B.1 – MAIN.C

```
1 #include <rv32i_io.h>
2 #include <utils_uart.h>
3
4 // -----
5 // -----
6 int main ()
7 {
8     // matrix data
9     int rowa = 100;
10    int cola = 100;
11    int rowb = 100;
12    int colb = 100;
13
14    int sum;
15    int Ma[rowa][cola];
16    int Mb[rowb][colb];
17    volatile int Mc[rowa][cola];
18    // float sum;
19    // float Ma[rowa][cola];
20    // float Mb[rowb][colb];
21    // volatile float Mc[rowa][cola];
22
23    // loop ctrl and others
24    int c, d, k;
25    int rem;
26
27    // timer
28    int t1;
29    int t2;
30    volatile int tf;
31
32    volatile char str[10] = "0123456789";
33
34    // get timer value
35    timer0_read(t1);
```

```
36
37 //-----
38 // M1 * M2
39 sum = 0;
40 if (cola != rowb){}
41 else
42 {
43     for (c = 0; c < rowa; c++)
44     {
45         for (d = 0; d < colb; d++)
46         {
47             for (k = 0; k < rowb; k++)
48             {
49                 sum = sum + Ma[c][k]*Mb[k][d];
50             }
51
52             Mc[c][d] = sum;
53             sum = 0;
54         }
55     }
56 }
57
58 // get timer value
59 timer0_read(t2);
60
61 //-----
62 tf = t2 - t1;
63
64 for (k = 0 ; k < sizeof(str) ; k++)
65 {
66     if (tf == 0)
67     {
68         str[k] = '\0';
69     }
70     else
71     {
72         rem = tf % 10;
73         tf /= 10;
74
75         str[k] = rem + '0';
76     }
```

```
77 }
78
79 for (k = sizeof(str)-1 ; k >= 0 ; k--)
80 {
81     if (str[k] != '\0')
82     {
83         uart_wpoll(str[k]);
84     }
85 }
86
87 // inf loop
88 while (1)
89 {
90 }
91 }
```