Implementação de um Processador RISC-V 32I em Verilog para Execução do Algoritmo MergeSort

Otavio Garcia de Oliveira Farias

July 2025

1 Algoritmo Escolhido

1.1 Análise no QTRVSIM

1.1.1 Algoritmos

Para realizar a escolha do algoritmo a ser utilizado, foram analisados na ferramenta QtRVSim, os algoritmos MergeSort, QuickSort e SelectionSort. O critério principal de escolha refere-se à quantidade de ciclos necessários para a execução do algoritmo.

- **MergeSort:** 333 ciclos + 32 stalls + 792 stalls de dados + 2990 stalls de programa = **4.147 ciclos**
- **InsertionSort:** 148 ciclos + 18 stalls + 400 stalls de dados + 1.270 stalls de programa = **1.836 ciclos**
- **QuickSort (recursivo):** 913 ciclos + 10 stalls + 1.186 stalls de dados + 8.180 stalls de programa = **10.289 ciclos**
- **QuickSort (iterativo):** 347 ciclos + 61 stalls + 852 stalls de dados + 2.830 stalls de programa = **4.090 ciclos**

A utilização do QuickSort recursivo apresentou alguns problemas, então foi utilizado um QuickSort iterativo encontrado na *web*. Os outros ordenadores foram gerados por uma compilação de um código em C.

Apesar do SelectionSort ter apresentado bom desempenho, não foi escolhido, pois, devido à sua complexidade $O(n^2)$, o seu comportamento para vetores maiores seria indesejável.

Outro ponto favorável à escolha do MergeSort é em relação ao tempo de execução constante; seu tempo de execução é constante, independente dos dados usados, algo que não ocorre no QuickSort.

1.1.2 Caches

Inicialmente foram analisadas diversas configurações de cache, variando parâmetros, como tamanho do bloco, tamanho da cache, política de substituição e grau de associatividade. Contudo, como uma primeira implementação de cache, foi decidido utilizar uma cache mais simples, com grau de associatividade um, de 128 blocos, com uma word (32 bits) por bloco.

Table 1: Desempenho da Cache de Instruções

Blocos da cache	**Hit**	**Miss**	**Memory Stall**	**Improved speed (%)**
1	32	299	2990	100
1	32	299	2990	100
2	179	152	3040	98
2	179	152	3040	98
4	254	77	3080	97
4	254	77	3080	97
8	292	39	3120	96
8	292	39	3120	96

Table 2: Desempenho da Cache de Dados

Blocos da cache	**Política**	**Hit**	**Miss**	**Memory Stall**	**Improved speed (%)**
1	write through	35	104	1391	92
1	writeback	35	104	1454	85
2	write through	59	47	1291	76
2	writeback	59	47	1390	69
4	write through	80	26	1391	71
4	writeback	80	26	1400	69
8	write through	97	9	1071	90
8	writeback	97	9	792	117

Como a utilização das caches representou uma melhora considerável no desempenho, foi decidida a utilização de caches para a memória de dados e de instruções, com auxílio de um componente decisor, para indicar qual cache deveria utilizar a RAM.

1.1.3 Branch Predictor

Na tentativa de diminuir o número de ciclos necessários para a execução, foi analisado o uso de um branch predictor nas suas configurações padrões do QtRVSim, para a ordenação de 50 elementos com o MergeSort. Sem o branch predictor, foram necessários cerca de 30.000 ciclos; com a utilização do BranchPredictor, foram economizados 500 ciclos. Uma diminuição de 1,67%, um valor baixo tendo em vista a complexidade da implementação, por isso, a ideia foi descartada.

1.2 Compilação do Código para Linguagem Assembly

Para gerar o código assembly, primeiramente foi utilizado um código em C do MergeSort, contendo um *main* simples, apenas chamando a função de ordenação, o código do ordenador foi o mesmo para todos os testes, apenas alterando o respectivo *main* para carregar o vetor de diferentes tamanhos. Durante a execução, todo o vetor é carregado do endereço da memória inicial e copiado para a pilha, onde será ordenado.

Para a compilação, foi utilizado o compilador $riscv64-unknown$, que gerava os códigos em hexadecimal; após isso, era necessário converter para binário.

Os seguintes comandos foram usados para a compilação:

Gerar código assembly a partir de C:
$riscv64 - unknown - elf - gcc - Sfile.c - ofile.s$
Montar o arquivo assembly:
$riscv64 - unknown - elf - asfile.s - ofile.o$
Desmontar para ver o assembly final:
$riscv64 - unknown - elf - objdump - dfile.o > final.s$

Observações: os desvios incondicionais não ficavam com o endereço correto, sendo necessária a modificação manual. As modificações em relação ao campo $funct3$ precisavam ser alteradas manualmente. Para facilitar no cálculo dos desvios e garantir que o programa comece a execução de maneira correta, no início do código, um $jalr$ era realizado para o *main*, que era sempre a última função do programa.

1.3 Informações Gerais do Código Gerado

Tamanho do código para ordenar um vetor de 1024 elementos: $(207 + 39) \times 4 = 984$ *bytes*

Tamanho dos dados: $1024 \times 4 = 4.096$ *bytes*

2 Análise das Instruções

2.1 Instruções Utilizadas

Foram implementadas apenas as instruções necessárias para a execução do algoritmo de ordenação.

sub srli slli or add srai
addi lui auipc
sw lw
blt bge jalr jal

A fim de simplificar a lógica da ULA, diminuindo a quantidade de *bits* necessária para a escolha da operação a ser realizada, as seguintes instruções tiveram seus campos $funct$ modificados:

Table 3: Instruções Modificadas

Instruções	$funct$ original	$funct$ modificado
sub	000	011
sra	101	010
bge	101	111

As modificações das instruções *sub* e *sra* foram realizadas para garantir que apenas com o $funct3$ seja possível utilizar todas as operações da ULA, isso foi possível pois nem todas as operações existentes na arquitetura RISC-V foram implementadas. Por exemplo, originalmente o $funct3$ das instruções *add* e *sub* é idêntico. Sendo assim, usado o $funct7$ para diferenciar qual foi a escolha.

2.2 Recorrência de instruções

2.2.1 MergeSort

Table 4: Contagem de instruções do código assembly do MergeSort

Tipo de Instrução	**Quantidade**
`addi`	67
`add`	8
`sub`	7
`sw`	34
`lw`	28
`srli`	5
`slli`	7
`or`	4
`blt`	6
`ble`	2
`bge`	3
`beq`	3
`jal`	2
`jalr`	1
Total	**207**

2.2.2 Main (usando como base configuração para 1024 elementos)

Table 5: Contagem de instruções do código assembly da função `main`

Instrução	Quantidade
`lui`	7
`addi`	9
`add`	5
`mv`	1
`lw`	6
`sw`	5
`bne`	1
`li`	2
`auipc`	1
`jalr`	1
`jal`	1
Total	**39**

3 Componentes do processador

3.1 Top

A unidade topo é responsável por controlar os sinais de *clock* e *reset* passados pelo *testbench* e instanciar os outros componentes, fazendo a ligação dos sinais *wire* entre componentes. Além de organizar todo e qualquer *mux* do pipeline.

3.2 Unidade de Controle

A unidade de controle decodifica o *opcode* e coordena os sinais que serão usados pelo pipeline. Os sinais implementados foram: EscReg, EscMem, ulaImm, jump, blt, bge, lui, aluControl, auiPc, jalr e lw.

Por serem utilizados em diversas instruções do ordenador, os sinais $ulaImm$ e $EscReg$ tiveram seus valores invertidos, sendo o valor lógico 0 interpretado como *verdadeiro*. Desse modo, economizando energia, pois ficariam com valor zero durante maior tempo de execução.

Algumas instruções como *srai*, utilizam o *shamt* para compor o seu valor imediato, como nas demais instruções o campo $funct7$, espaço da instrução equivalente ao *shamt*, é sempre zero, não foi utilizado o *shamt*, pois seu valor não alteraria nada durante a execução.

As instruções de desvio condicionais (*blt* e *bge*) utilizam a mesma operação da ULA, portanto, para ambos os casos o aluControl será 100, a diferença deles será no momento da verificação durante o estágio MEM, utilizando o *bit* menos significativo da saída da ULA.

3.3 Decodificador de Imediato

O imediato das instruções no RISC-V não apresenta um comportamento uniforme, portanto, a unidade de controle recebe a instrução e, através do opcode, concatena os bits necessários para formar o imediato no padrão desejado. Como desvios não podem ser feitos para valores ímpares, a memória é separada em *bytes*, o *bit* menos significativo será sempre zero nas instruções de desvio.

3.4 Banco de Registradores

O banco de registradores controla o acesso aos 32 registradores. O "registrador" *zero* sempre terá o valor zero, portanto, foi utilizada uma constante com esse valor ao invés de um registrador. Os demais registradores funcionam normalmente, os dados são lidos independentemente do clock e escritos na borda do clock.

3.5 ULA

A Unidade Lógica Aritmética (ULA) é responsável por realizar os cálculos e decidir se o desvio de algum *branch* será realizado. A codificação de qual operação será realizada foi feita apenas com o $funct3$, pois foram usadas menos de três operações na ULA. Portanto, algumas instruções tiveram sua codificação modificada.

4 Implementação do Pipeline

4.1 Registradores de Pipeline

O *pipeline* foi dividido entre cinco estágios, em cada estágio encontram-se registradores para armazenar os dados necessários para o próximo estágio.

IF: Busca da instrução; ID: Decodificação da instrução e do imediato, acesso para busca de dados no banco de registradores; EX: Cálculos na ULA; MEM: Acesso à memória de dados e decisão dos desvios; WB: Escrita no banco de registradores.

4.2 Hazard de Desvio

Quando um desvio é tomado no estágio mem, um sinal é enviado para a memória de instruções, zerando a próxima instrução, e para os registradores de pipeline dos estágios IF_ID, ID_EX, EX_MEM, eles realizam um $flush$, zerando todos os sinais, desse modo nada será executado no pipeline, e no próximo ciclo, a nova instrução já será executada.

4.3 Forwarding

Um componente separado é responsável por detectar e tratar os hazards de dados, que ocorrem devido a utilização do pipeline. Os dados são desviados ao longo do pipeline para a ULA e para o endereço da instrução SW.

Para a detecção, a unidade de forwarding analisa se um dado, que será utilizado pela ULA ou como endereço para um SW, será salvo por alguma instrução que está à frente no pipeline. Caso isso ocorra, serão enviados sinais para decidir qual dado será usado para alguns mux distribuídos ao longo do pipeline, que pegam dados do caminho normal do pipeline, saída da ULA, saída da memória de dados e do estágio WB.

Como os dados precisam estar prontos no estágio EX, precisa-se tratar os dados a serem salvos que estão em MEM e WB, Contudo, se o desvio for feito para o estágio EX, caso alguma instrução tenha salvo no estágio WB anterior, o dado precisará ser buscado no banco de registradores. Com o objetivo de contornar esse problema, desvia-se os dados dos estágios EX, MEM, WB para a entrada dos registradores de pipeline entre ID_EX.

Nesse sentido, um novo problema surge: se uma instrução que esteja no estágio ID, onde será identificado o forwarding, precise de um dado vindo de um lw no estágio MEM, o dado somente estará pronto no próximo ciclo. Desse modo, um mux foi colocado no estágio EX, pegando um valor que no ciclo anterior estava saindo da memória de dados, e agora está no estágio WB para ser usado.

Portanto, a unidade de forwarding está dividindo entre os estágios ID e EX, para tratar todos os hazards de dados presentes no pipeline.

5 Implementação das Memórias Caches

5.1 Lógica das Caches

Foram implementadas duas caches, uma para memória de instruções e outra para a memória de dados, ambas com grau de associatividade um, de 128 blocos, com uma word (32 bits) por bloco.

A ideia utilizada foi de estágios para a cache, que são divididos através do uso de *case*, o *index* representa os 7 *bits* menos significativos e os demais representam a *tag*. Um vetor para a validade dos dados e um vetor de *tags*, ambos indexados pelo *index*.

Caso os *bits* mais significativos do endereço coincidam com a *tag* do vetor de *tags*, ocorre um *hit*. Desse modo, os dados são acessados diretamente; caso ocorra um *miss*, o estado da memória é modificado para acessar a RAM. Na memória de instruções, a cache é atualizada com dados da RAM e passa para o estágio inicial; nesse caso, ocorrerá um *hit* e os dados serão usados de forma correta.

Na memória de dados, existe um vetor de *dirty*, indicando se o dado foi sobrescrito na cache após ter sido retirado da RAM. Então, em caso de *miss*, primeiro é verificado se o dado está marcado como *dirty*, caso esteja, a memória

passa para o estado *write_back*, onde acessará a *RAM* e escreverá o dado antigo; posteriormente, passará para o estado de atualizar a cache e seguir com o uso normal.

Ambas as caches em caso de *miss* emitem um sinal para um componente *memories* que controla o uso das memórias, que avisa o pipeline se é necessário parar e esperar. Dentro das caches, quando é necessário utilizar a *RAM*, a cache espera um sinal, indicando que a *RAM* já processou os dados; caso contrário, a cache permanece no mesmo estado.

5.2 Decisor

Um componente chamado *CacheToRAM* decide qual cache utilizará a *RAM* caso ambas façam uma requisição ao mesmo tempo, emitindo um sinal de espera caso a operação na *RAM* ainda não esteja concluída, seja pela demora natural do acesso ou seja pela utilização da *RAM* pela outra cache. De maneira arbitrária, foi decidido que a prioridade de acesso será para a cache de instruções.

5.3 Espera do pipeline pela cache

Caso o componente *memories* indique que é necessário parar, o pipeline para de enviar o sinal de *clock* para os componentes, exceto as caches. Para isso, foi utilizado um *realClock* controlado pelo *testbench* e um *clock* utilizado pelo pipeline. Em caso de *hit* o *clock* recebe o valor do *realClock*; em caso de algum *miss*, o *clock* recebe seu próprio valor, não atualizando os componentes do pipeline.

5.4 Dados das caches implementadas

Para analisar o comportamento da cache, foram colocados registradores no *Top* para contar o número de ciclos e nas caches para contar a quantidade de acessos e hits.

Para a análise da melhoria será usada uma penalidade de acesso a *RAM* arbitrária de 4 ciclos, um hit time de 1 ciclo e a seguinte fórmula:

$$Speedup = \frac{Tempo\ sem\ cache}{AMAT} = \frac{Misspenalty}{Hittime + (Missrate \times Misspenalty)}$$

5.4.1 4 elementos

Ciclos: 3.744
- Miss Instruções: 1.592
- Acessos Cache Dados: 1.332
- Miss Dados: 145
- Dirty: 87

$$MissRate_{instr} = 41,52\% \quad \Rightarrow \quad Speedup_{instr} \approx 1.5\times$$

$$MissRate_{dados} = 10.88\% \quad \Rightarrow \quad Speedup_{dados} \approx 2.78\times$$

5.4.2 1024 elementos

Ciclos: 1.913.797
Miss Instruções: 665.640
Acessos Cache Dados: 770.978
Miss Dados: 89.798
Dirty: 68.848

$$MissRate_{instr} = 34.78\% \quad \Rightarrow \quad Speedup_{instr} \approx 1.67\times$$

$$MissRate_{dados} = 11.64\% \quad \Rightarrow \quad Speedup_{dados} \approx 2.72\times$$

6 Análise de métricas do processador utilizando a ferramenta Genus

Resumo de Células

Table 6: Resumo geral do módulo Top

Instance Module	Cell Count	Cell Area	Net Area	Total Area
Top	1843	9954.508	2005.177	11959.686

Resumo por Tipo de Célula

Table 7: Distribuição de células, área e potência de fuga

Type	Instances	Area	Leakage Power (nW)	Leakage Power (%)
sequential	832	5296.212	175.932	36.5
inverter	36	81.738	0.8	1.6
clock_gating_integrated_cell	32	2.1	5.984	1.2
logic	943	4368.622	292.623	60.7
total	**1843**	**9954.508**	**100.0**	**482.133**

Relatório de Temporização

Frequência atingida: 1,15GHz

Path 1: MET (0 ps) Setup Check with Pin `Mem_MemInst_stage_reg[0]/CK->D`
Group: `C2C`
Startpoint: (R) `Mem_MemInst_tagArray_reg[44][5]/CK`
Clock: (R) `realClock`
Endpoint: (F) `Mem_MemInst_stage_reg[0]/D`
Clock: (R) `realClock`

Table 8: Análise de Setup

	Capture	**Launch**
Clock Edge	870	0
Src Latency	0	0
Net Latency	0 (I)	0 (I)
Arrival	870	0
Setup		46
Required Time		824
Launch Clock		0
Data Path		824
Slack		0

References

[1] Jakub Dupak, Pavel Pisa, Martin Stepanovsky, and Karel Koci. Qtrvsim – risc-v simulator for computer architectures classes. In *Embedded World Conference 2022*, pages 775–778, Haar, 2022. WEKA FACHMEDIEN GmbH.

[2] RISC-V INTERNATIONAL, S. l. *The RISC-V Instruction Set Manual: Volume I – Unprivileged Architecture*, 2025. Version 20250508.