Relatório
Laboratório 3: Processador RISC-V Uniciclo

Grupo 3
Gabriel Alves Castro - 17/0033813
Henrique Mendes de Freitas Mariano - 17/0012280
Luthiery Costa Cavalcante - 17/0040631
Matheus Breder Branquinho Nogueira - 17/0018997

Universidade de Brasília - Dep. Ciência da Computação
Disciplina CIC 116394 - Organização e Arquitetura de Computadores - Turma A

Introdução

O projeto dessa quinzena é sintetizar no software Quartus Lite o processador RISC-V Uniciclo, com a ISA base RV32I de instruções de inteiros, acesso à memória e desvios condicionais e incondicionais, e em seguida com sua extensão RV32IM com instruções aritméticas de multiplicação, divisão e resto. É o primeiro de 3 projetos cumulativos, seguido das implementações Multiciclo e Pipeline.

Solução

Dificuldades?... Por onde começar?

Antes de falar do desenho do datapath em si, a maior dificuldade inicialmente era entender como ele estava representado no código em verilog. "Onde está esse fio? O que isso tá fazendo? `DwWriteEnable`?? Não vejo isso no diagrama!".

Trata-se de um projeto bem extenso, desenvolvido ao longo de dezenas de semestres, com tratamento sofisticado de exceções, interface de áudio e vídeo, etc... que deixa o datapath, por exemplo, encharcado de entradas, saídas e sinais de monitoramento. Saber o que podia somente retirar e o que iria bagunçar inteiramente o programa foi um obstáculo enfrentado. Nessa parte entendemos mais ou menos a hierarquia do projeto (topDE - CPU - Datapath).

Uma das primeiras conclusões que tivemos sobre o datapath em si foi copiar a ULA do laboratório anterior, já implementada para RISC-V, para o projeto, em vez de editar a original, do MIPS. Vimos nela instruções úteis como o `lui`. Falaremos dele já já.

A iniciativa inicial para fazer esse projeto andar foi projetar a tabela verdade do bloco de controle geral e do controle da ULA numa planilha. Aqui tomamos decisões de projeto importantes, como decidir se o `lui` seria feito pela ULA ou conectado diretamente no multiplexador `Mem2Reg`, quantos bits teriam essas seleções (algumas tinham mais bits e mais casos possíveis, em relação

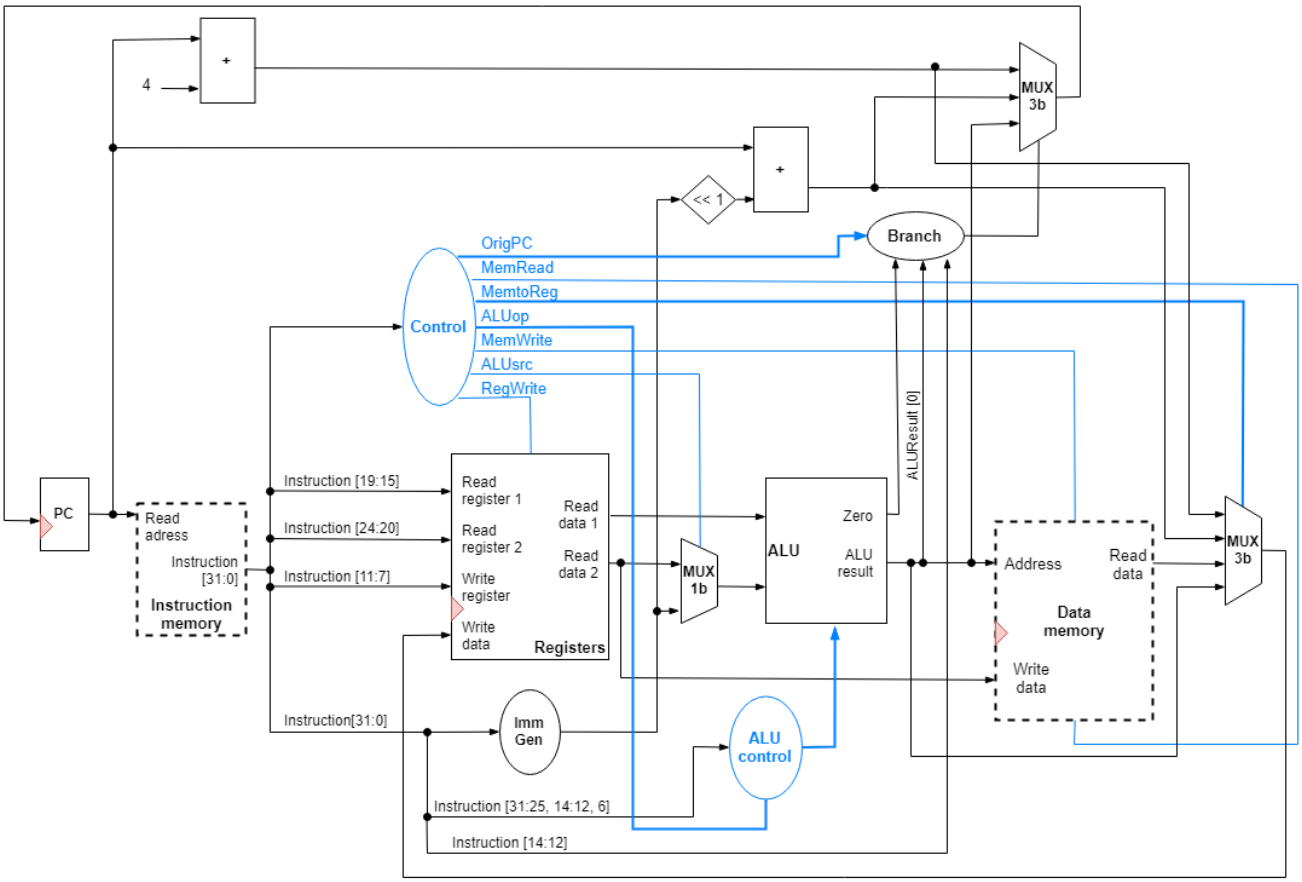

Figura 1: Diagrama do datapath uniciclo.

à implementação do diagrama), Como implementar o jalr que joga o resultado da ULA no PC, e como funcionariam os branches.

No MIPS, há somente duas instruções de desvio condicional nativas, o `beq` e o `bne`, cujos resultados podem ser avaliados apenas com a saída `oZero` da ULA. Porém, no RISC-V, temos mais 4 tipos de branches, onde esse dado é inconclusivo. Por sorte temos instruções SLT e SGE na ULA. Podemos assim verificar a condição com o próprio resultado da operação. O controle de branch ganhou mais uma entrada e ficou um pouco mais complicado que um simples AND. Não complexo o bastante pra ganhar um bloco separado como o ImmGen, contudo.

Isso muda também o controle da ULA. Era dito que em todo branch seria feita uma subtração (01) na ULA. Agora precisamos diferenciar o caso beq/bne dos outros pelo funct3. O caso 10 ficou com as intruções tipo R e o caso 11 com o tipo I; foi necessário separá-las por não poder se diferenciarem pelo funct3. O auipc e o jal, que não usam a ULA (don't care), além dos acessos à memória e o jalr, foram pro 00. Sobrou uma só instrução: o lui, que não tem funct3 nem funct7 e nem faz operação add na ULA. Foi preciso enfiá-la em uma das opções e incluir mais uma entrada - o bit 6 do opcode - para que se pudesse diferenciar. Foi pra junto dos branches.

Depois do alívio de só precisar alterar alguns arquivos do projeto, vimos nos blocos de memória a parte mais nebulosa do código. Nunca sabíamos o que era e o que não era preciso mudar. Sabíamos, contudo, que a memória de código/dados do sistema e do usuário virou uma só, com a ausência dos setores kdata e ktext do MIPS.

Tabelas-verdade estão na planilha dentro da pasta principal do projeto.

Requerimentos Físicos e Temporais

Tabela 1: Requerimentos Físicos sem o módulo de multiplicação.

ALMs	**Num. Reg.**	**Qt. bits de mem.**	**DSP**
3,700 / 32,070	4,287	934,912 / 4,065,280	12 / 87

Tabela 2: Requerimentos Temporais sem o módulo de multiplicação.

Tpd	**Th**	**Tco**	**Tsu**	**Su Sla.**	**Hd Sla.**	**Rec. Sla.**	**Rem. Sla.**	**Mín. Pul**	**Máx.Freq.**
25.639	3.781	23.162	33.659	13.333	0.025	31.119	0.359	15.007	40.0MHz

Tabela 3: Requerimentos Físicos com o módulo de multiplicação.

ALMs	**Num. Reg.**	**Qt. bits de mem.**	**DSP**
6,464 / 32,070	4,279	934,912 / 4,065,280	21 / 87

Tabela 4: Requerimentos Temporais com o módulo de multiplicação.

Tpd	**Th**	**Tco**	**Tsu**	**Su Sla.**	**Hd Sla.**	**Rec. Sla.**	**Rem. Sla.**	**Mín. Pul**	**Máx.Freq**
22.624	3.730	22.664	129.127	13.278	0.127	30.670	0.280	15.015	40.0MHz

testbench.s

O testbench.s encontra-se na pasta "Docs/testes".

testeECALLv1.s

Para podermos executar o testeECALLv1.s tivemos que fazer mudanças nos programas testeECALLv1.s e SYSTEMv1.s, essas mudanças foram: a troca do uret por ret no SYSTEMv1.s e ecall por jal exceptionHandling e ret no testeECALLv1.s, mas essa mudança só funcionava até o primeiro jal CLS onde a tela é pintada de vermelho, pois o endereço do registrador ra não estava sendo salvo com isso o ret sempre retornava para ele mesmo, a partir disso tivemos que guardar o valor do registrador ra na pilha antes de entrar no jal exceptionHandling e depois que o exceptionHandling retoma ao código principal nós recuperamos novamente da pilha o valor correto do registrador ra esses foram as maiores dificuldades que encontramos para poder executar o testeECALLv1.s. Porém o testeECALLv1.s não está funcionando totalmente correto, pois a leitura do inteiro não está funcionando corretamente. O funcionamento do testeECALLv1.s encontra-se neste link.