Microprocessador Ciclo Único

Juliano Mourão Vieira

21 de outubro de 2024

Sumário

Nota Sobre Este Material

Isto aqui foi feito a toque de caixa para a transição da disciplina para o processador RISC-V, portanto algumas coisas não estão tão bem finalizadas. Favor comunicar erros e sugestões ao professor. (Em especial, desculpem pelos desenhos da ULA e da AND...)

---

1 Princípios

Precisamos estabelecer o funcionamento básico de um microprocessador, mas você *já sabe isso*. O microprocessador é o circuito que executa um programa. O programa a ser executado é uma sequência de instruções em código de máquina, e estas instruções precisam estar armazenadas em alguma memória, em sequência.

1.1 O que o Processador Faz?

A instrução que o microprocessador vai executar precisa ser lida da memória; o contador de programa (PC – Program Counter) é o registrador que vai indicar a posição de memória em que ela está.

Uma vez lida a memória de programa, a instrução é então decodificada. *Decodificar uma instrução* significa identificar qual instrução ela é (soma, cópia de valores, condicional etc.) e gerar os sinais para o restante do circuito executá-la.

Uma das instruções mais simples é a soma de dois valores, realizada por uma Unidade Lógico-Aritmética, ou ULA. Os valores somados vêm dos registradores internos do processador, que agem como nossas variáveis em programação, ou vêm da própria instrução. A operação de soma vai ser selecionada pelo bloco de decodificação.

O diagrama em blocos da figura 1 mostra isso.

Ele está em esboço porque ainda estamos pensando a respeito. Observe as siglas, em especial, “ROM”, uma memória apenas de leitura, que é um

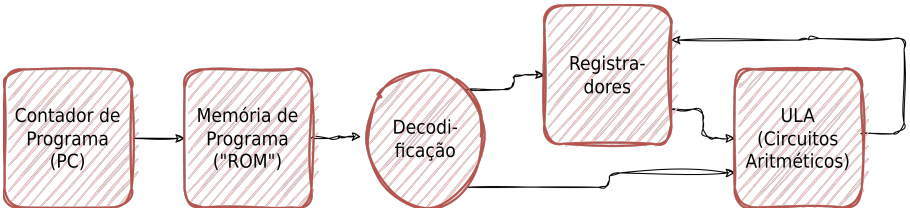

Figura 1: Esboço do microprocessador

anacronismo que eu uso para simplificar o diagrama. Note que a memória de programa não vai ser alterada depois de o programa ser carregado nela (exceto em casos bizarros de código automodificável) por isso usar uma ROM faz sentido.

1.2 Como Fazer Funcionar?

A ordem dos eventos no nosso processador vai ser a seguinte:

1. O microprocessador lê a instrução da ROM; o endereço é indicado pelo PC.
2. A instrução é então *decodificada*. Decodificar significa identificar qual é esta instrução e produzir os sinais de controle adequados.
3. Os operandos são acessados e passados à frente, ou seja, os dados da instrução são determinados.
4. O bloco responsável pela instrução determina seu resultado.
5. O resultado da instrução é finalmente escrito no seu registrador destino.

Tudo isso deve acontecer em um único clock (no capítulo seguinte usaremos vários clocks por instrução). Necessariamente, precisamos de duas memórias, uma ROM para o código e uma RAM para os dados, pois *ambas estarão ativas no mesmo clock,* o que significa que esta é uma arquitetura Harvard.

Claro que essa sequência é apenas a típica; outros tipos de instruções vão necessitar de variações nos passos.

1.3 Nomeando os Sinais

Vamos agora incluir a memória RAM de dados, como visto na figura 2, aproveitando para nomear os sinais principais (todos eles são barramentos, ou seja, conjuntos de vários bits).

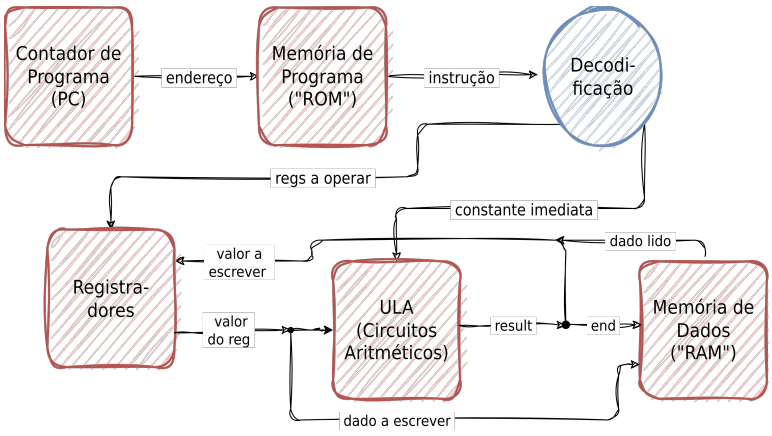

Figura 2: Esboço com os nomes dos sinais principais

O endereço da RAM que vai ser lido ou escrito é calculado pela ULA pois as instruções sempre exigem um ponteiro e um deslocamento constante a ser somado a ele. O dado a escrever na RAM (pela instrução sw) vem de um registrador e o dado lido da RAM (por uma instrução lw) deve ser escrito em um registrador, obrigatoriamente.

Note enfim que há um curto-circuito neste rascunho; iremos resolvê-lo mais adiante com um mux.

2 Background Necessário de Eletrônica Digital

Não iremos precisar de nada difícil. Embora em projetos profissionais sejam utilizados métodos, técnicas e ferramentas avançados, aqui precisamos só do arroz-com-feijão para um circuito simples e pé-no-chão.

2.1 Registrador

É isso mesmo que você está pensando: um registrador é só um amontoado de n flip-flops D em paralelo com o clock em comum, ou seja, um circuito que guarda um número, ou uma memória de n bits. Portanto é só um componente que armazena um número.

O mais importante a saber de um registrador é quantos bits ele pode armazenar, o que determina o maior e o menor número representável. O RISC-V possui um PC de 32 bits, o que implica que a ROM pode ser acessada do endereço 0 até cerca de 2^{32}, o que nos dá 4 GBytes. (Um endereço, por default, não é sinalizado – é *unsigned*).

Também é importante notar que normalmente usamos registradores com a escrita sincronizada na borda de subida do pino de clock e apenas quando o pino de *write_enable* estiver ativo; caso contrário os dados de entrada são ignorados.

2.2 Multiplexador

Um multiplexador (*multiplexer* se você não adivinhou) ou mux é apenas um seletor de dados. Na sua forma mais simples, temos duas entradas e vamos escolher uma delas como resultado, usando um pino de seleção. Então o funcionamento dele é assim:

$$\text{saída} = \begin{cases} \text{entrada0}, & \text{se seleção} = 0 \\ \text{entrada1}, & \text{se seleção} = 1 \end{cases}$$

É como se fosse um interruptor controlado pelo pino de seleção, "curto-circuitando" uma das entradas até a saída: se seleção = 1, tudo que acontecer em entrada1 será visto na saída do mux, e o outro caso é análogo.

É só isso.

2.3 Demais Assuntos

Uma ULA (Unidade Lógico-Aritmética) é um circuito que faz contas entre dois números; uma terceira entrada faz a seleção de qual operação será realizada. Além do resultado, o *status* da operação também é indicado por saídas conhecidas como *flags* (sinalizadores), como *carry* ou *overflow*.

Circuitos sequenciais são aqueles que possuem um estado interno, determinado por alguns valores armazenados em flip-flops. Vamos nos restringir a máquinas de estado com transições bastante simples, sem necessitar de nenhum método formal. (*Spoiler:* o nosso estado é dado essencialmente pelo PC e registradores do banco, mais as flags).

E espera-se naturalidade de trabalhar com binário, claro.

3 Como funciona?

Agora que nosso esquema está um pouco melhor, vamos desenhá-lo sem usar o rascunho, na figura 3; já podemos chamar o bloco de decodificação pelo nome verdadeiro, Unidade de Controle. Os nomes dos sinais estão indicados com linhas pontilhadas verdes.

A Unidade de Controle irá determinar, a partir da instrução, quais registradores devem ser operados, repassando isso ao Banco de Registradores. A seta sombreada com uma interrogação indica que há muitas funções que ainda não especificamos sendo feitas pela UC. Já está incluído um sinal azul de controle que identifica a operação que a ULA deverá fazer, que depende da instrução a executar.

Observe que a constante incluída na instrução tem tamanho variável conforme o formato utilizado (I, B, S,...). Isso deverá ser trabalhado, para a ULA poder operar com 32 bits.

Por fim, os dois curto-circuitos ainda a resolver estão indicados pelos círculos vermelhos.

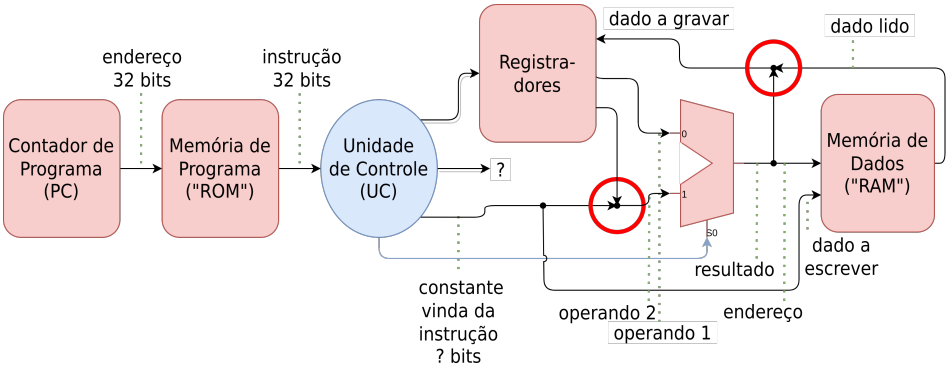

Figura 3: Esquemático inicial, com erros ainda

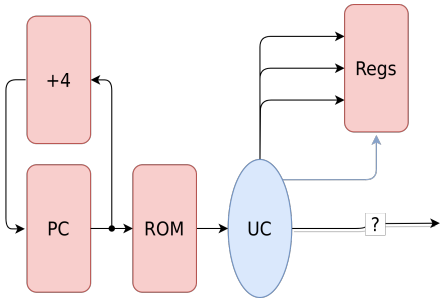

Figura 4: Diagrama do microprocessador – incremento do PC

3.1 WTF é um PC?

O *Program Counter* (Contador de Programa) é apenas um registrador. Ele guarda o endereço da próxima instrução a ser executada. Ele é chamado de "contador" por questões históricas (yes, dependência de caminho novamente!), pois originalmente havia uma instrução por endereço de memória de programa e a todo clock ele era incrementado (exceto por desvios). Hoje em dia ele é apenas um registrador; a atualização dele (um incremento ou jump ou qualquer outra coisa) é realizada por um circuito externo, talvez na UC. Como cada instrução ocupa 4 endereços[1], precisamos *somar 4 ao PC* para avançar para a próxima instrução.

Agora ficamos como na figura 4, usando siglas para despoluir o diagrama.

3.2 WTF é uma UC?

A Unidade de Controle é o circuito do processador responsável por identificar a instrução que vai ser executada e gerar os sinais de controle para executá-la. Este sinais de controle irão conduzir os valores relevantes pelos caminhos

[1]Cada endereço de memória tem um byte, cada instrução tem 4 bytes = 32 bits, portanto cada instrução ocupa 4 endereços de memória.

do circuito que irão fazer a execução em si da instrução.

Na prática, os sinais de controle normalmente são sinais binários que funcionam como seletores e como chaves liga/desliga. Por exemplo, se a instrução a ser executada é `add s1,s2,s3`, a escrita em um registrador deve ser habilitada, o acesso à memória de dados deve ser desabilitado (pois ela não é usada), o resultado a ser gravado é selecionado vindo da ULA (ao invés de outras fontes) e assim por diante.

Para gerar este sinais de controle, a UC tem um circuito combinacional simples que, de acordo com o opcode (e os campos funct) da instrução, identifica quais instruções vão exigir os sinais de controle. Por exemplo, a instrução `addi s4,zero,123` também vai gerar escrita de registrador com o resultado vindo da ULA (e RAM desabilitada), mas um dos operandos da ULA vem de uma constante imediata da instrução ao invés de vir de um registrador; portanto, um mux seletor é necessário para fazer essa escolha, diferenciando add de addi.

3.3 Como Funfa um Banco de Registradores?

Processadores modernos possuem vários registradores de uso geral, com instruções normalmente ortogonais; isso implica que o acesso a eles deve ser simples de identificar a partir dos bits da instrução.

A maneira mais fácil de se entender um banco de registradores é pensar nele como se fosse uma memória de dados, com cada registrador tendo seu "endereço" identificador. Desta forma, iremos atribuir um número a cada um dos 32 registradores do RISC-V, e este número escolhe qual registrador será lido ou escrito. O registrador em si é apenas uma coleção de flip-flops D em paralelo, como visto na disciplina de Circuitos Digitais, com um clock para sincronização, que por conveniência vamos assumir sendo de rampa de subida, e com um pino de *Write_Enable* ou *wr_en* que, quando em nível alto, realiza a escrita dos dados de entrada no registrador.

O banco é então apenas um bloco que permite o acesso a estes registradores internos. Ele é capaz de ler dois registradores simultaneamente e escrever em um terceiro registrador qualquer; pense numa instrução como `add s1,s2,s3`, que faz a leitura de s2 e s3 simultaneamente, e escreve o resultado em s1. Confira a pinagem na figura 5.

Dois registradores serão lidos simultaneamente: o número deles é colocado em reg_r1 e reg_r2 (de *register read,* registrador a ser lido) e o valor lido é apresentado nos pinos data_r1 e data_r2, respectivamente. Para a escrita, colocamos o dado nos pinos data_wr (*data to write,* dado a escrever), especificamos o registrador a ser escrito através de reg_wr (de *register write*) e habilitamos a escrita setando wr_en (*write enable*) para 1; na próxima rampa de subida do clock, a escrita será realizada.

Se consultarmos as tabelas de instruções do RISC-V (podem ser as do laboratório de assembly ou da tabela de instruções completa), veremos os

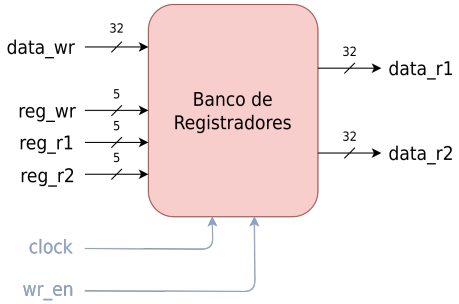

Figura 5: Bloco do Banco de Registradores – os dados são barramentos de 32 bits e a seleção dos registradores precisa de 5 bits cada

números atribuídos aos registradores que estão nomeados na instrução, precedidos por uma letra x (note estes números são decimais, o x indica que é um registrador[2]). Assim, para a instrução anterior, teremos o x18 identificando s2, x19 para s3 e x9 para s1; quando ela for executada, o número $18_{10} = 10010_2$ será escrito em reg_r1 e o número $19_{10} = 10011_2$ em reg_r2, e o número $9_{10} = 01001_2$ será escrito em reg_wr, identificando deste modo os três registradores que serão operados.

O dado a ser escrito será fornecido pela ULA (pois a instrução add é uma soma), o clock é o clock global do microprocessador (será um clock único para todos os componentes que precisam de clock) e o sinal de wr_en, que identifica o momento em que a escrita irá ocorrer, será gerado pela Unidade de Controle, como visto na seção 5.

4 O que Vamos Implementar Aqui?

Este processador é apenas didático, uma simplificação da já bastante simples ISA RV32I do RISC-V. Iremos ilustrar os comportamentos e implementações básicas para um processador ser útil o suficiente, Turing completo e de forma que demais instruções básicas tenham circuitos simples de se intuir.

4.1 Instruções Implementadas

Portanto as instruções que nos interessam e serão implementadas são:

- add;
- addi;
- beq;
- lw;

[2]Eu sei, eu sei, péssima ideia. *Tech nerds* são pessoas péssimas para nomear coisas.

- sw.

Certas instruções são triviais de se adicionar (sub é como add, mas a ULA faz subtração; bne é como beq, mas o sinal da comparação é invertido); outras não alteram muito o circuito, mas adicionam alguma complexidade pontual (p. ex., os *branches* blt e bltu exigem circuito comparador de dois números com a função menor, nos sabores *signed* e *unsigned,* respec.). Os desvios incondicionais são simples de se simular (com algo como `beq zero, zero, destino`), embora o montador utilize outra instrução.

4.2 Casos Omitidos

Dois casos principais são omitidos aqui. O primeiro são as instruções de chamada de função e retorno (jal e jalr) que adicionariam complicações na decodificação e na compreensão dos mecanismos pelo aluno.

O segundo diz respeito à determinação das constantes em outros formatos de instrução que decidimos ignorar (S, U e J); no total, temos 5 formatos com diferentes especificações da constante envolvida na instrução, mas julgo que isso não traz muita novidade, já que são só embaralhamentos diferentes dos bits de constantes usando 3 larguras diferentes, ou seja, trabalho mais braçal do que intelectual.

Mesmo assim, cabe notar que instruções importantíssimas utilizam estes formatos, incluindo lui (que nos proporciona o uso de constantes de 32 bits no assembly) e o sw, cuja decodificação é vista em detalhes na seção 5 porém sem mencionar a forma de obter a constante. Mas acho que não faz falta.

A circuitaria também ficaria bem mais complexa se usássemos extensões da ISA, possivelmente gerando opcodes adicionais para identificar os formatos. Em especial, se usarmos a extensão RV32C de instruções comprimidas em 16 bits, o circuito precisaria ser mais sofisticado para diferenciar as instruções normais das comprimidas e proceder de acordo.

5 Decodificação das Instruções

Pense na memória RAM de dados e suas duas operações possíveis: ou ela faz a leitura de um dado em certo endereço, ou ela escreve um dado em um endereço, caso contrário está inativa (em *idle,* sem fazer nada). Para indicar que a operação de leitura deve ocorrer, há um pino correspondente que deve ser levado a nível 1 e, para a escrita há outro pino com funcionamento análogo.

Digamos então que, para escrever um dado em uma RAM, é necessário ativarmos tal pino, que podemos chamar de MemWrEn (*Memory Write Enable*), de algum jeito. Este sinal de controle deve ser gerado por um circuito, e só deve ser habilitado (ou seja, setar para nível 1) quando a instrução `sw`

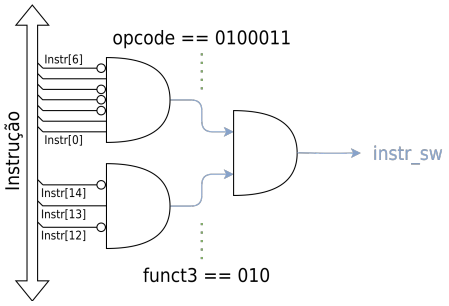

Figura 6: Decodificação de uma instrução sw

for executada, pois é a única que escreve na memória de dados; o sinal deve permanecer em 0 em todos os outros momentos.

5.1 Identificando a Instrução

O código de máquina da instrução de 32 bits será lido da memória e os bits do opcode (os 7 bits LSB) irão identificar o grupo de instruções respectivo. Normalmente, precisamos comparar os bits do campo funct3 para identificar a instrução específica que será executada.

No caso da instrução `sw`, o opcode será 0100011_2 e o funct3 será 010_2, segundo a tabela do RISC-V, portanto qualquer código de máquina com estes bits será reconhecido como `sw`.

5.2 Gerando o Sinal

Primeiramente devemos identificar a instrução pelo opcode e demais campos. Seguindo a tabela, a comparação que teremos que fazer é:

```
instr_sw <= (instr[6..0]==0100011) AND (instr[14..12]==010)
```

Isso pode ser implementado com as portas lógicas da figura 6. O sinal gerado ("instr_sw") vai estar em nível 1 quando o código for um `sw` e em nível 0 para qualquer outra instrução.

A este procedimento de identificar a instrução pelos bits do código de máquina chamamos de *decodificação da instrução*.

5.3 Efeitos de um Sinal de Controle

A decodificação apenas gera um sinal booleano, sendo o primeiro passo para execução da instrução. Este sinal irá habilitar e selecionar determinados blocos no circuito do processador para que ele realize as operações necessárias. A circuitaria que gera estes sinais é chamada de Unidade de Controle.

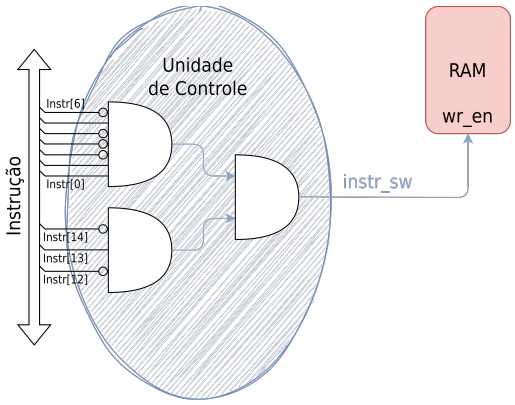

Figura 7: Circuito de controle para habilitação de escrita na RAM

Enables

O uso mais imediato do sinal `instr_sw` é habilitar ou desabilitar partes do circuitos, através de um pino de *enable* que o bloco em questão possui. Este pino habilita a função quando estiver em 1 e desabilita em 0.[3]

No nosso exemplo, considere a memória de dados RAM: ela deve ser habilitada para escrita no *write_enable* ou *wr_en* e, se houver, desabilitada para leitura no *read_enable* ou *rd_en* e habilitada para uso com um *enable* geral, já que a operação apenas vai escrever um dado, como se vê na figura 7.

Da mesma forma, apenas a instrução `lw` vai fazer a leitura da RAM, gerando o read_enable em 1. Todas as outras instruções vão deixar estes enables da RAM em nível 0, já que ela é desnecessária para estes casos.

Seletores para os Muxes

A outra forma de controlar as operações do microprocessador é escolhendo os dados a serem passados para determinados blocos, o que é feito trivialmente com multiplexadores. Vamos escolher um ponto chave no circuito para exemplificar a seguir: a seleção do segundo operando da ULA. Mas multiplexadores e seus sinais seletores estão espalhados por todo o microprocessador.

O primeiro operando é sempre um registrador e o segundo pode ser um registrador ou uma constante: pense na instrução `add s1,s2,s3` e na instrução `addi s1,s2,51`. Como fazer esta distinção? Obviamente, precisamos verificar os opcodes e campos funct. (Iremos simplificar bastante as coisas, assumindo que outras instruções não serão implementadas, mas no final das contas, é tudo porta lógica mesmo).

[3] A isto se chama *lógica positiva;* podemos usar o inverso, a lógica negativa, mas é incomum.

5.4 Diferenciando Formatos R e I

Uma instrução pode ser determinada de acordo com o seu opcode e, complementarmente, com os campos funct3 e funct7. Todos os casos possíveis devem ser cobertos, mas aqui vamos fazer uma simplificação razoável: ao nos restringirmos ao core ISA RV32I do RISC-V, basta verificarmos se o opcode é 0110011_2 para ter certeza de que se trata de uma instrução R. Neste caso, a ULA irá utilizar dois registradores para fazer a conta, independentemente dos valores de funct3 e funct7.

No caso do formato I, iremos obter uma constante a partir do código da instrução, e esta vai para a ULA em uma conta com um dos registradores. As instruções I podem ser reconhecidas pelo opcode 0010011_2 desprezando funct3 no RV32I. Circuitos similares aos utilizados na seção 5 poderão gerar um sinal que diferencia os formatos.

Para demais formatos e extensões da ISA, o processo é bastante semelhante.

Constantes e Extensão de Sinal

Para as instruções `addi`, usa-se o formato I (de *constante imediata*), como visto no laboratório 2 de assembly. Neste formato, os 12 bits MSB representam uma constante sinalizada com alcance dado por $-2^{11} \leq cte \leq 2^{11} - 1$. Para podermos realizar contas na nossa ULA de 32 bits, é necessário converter este número de 12 bits para um número de 32 bits.

Para realizar esta assim chamada *extensão de sinal,* basta observar o bit MSB da constante, ou seja, o bit b11 dela, que corresponde ao bit b31 da instrução, o MSB, mais à esquerda. Seguindo as regras de complemento de 2, números positivos terão este bit em 0 e números negativos terão obrigatoriamente este bit em 1.

Este valor de sinal é copiado 20 vezes e concatenado à esquerda da constante de 12 bits, nos dando uma nova palavra de 32 bits que possui o mesmo valor da constante da instrução. Para fazermos isso, no circuito, basta derivar 20 novos fios, curto-circuitados com o MSB da constante original

5.5 Enfim o Mux!

Vamos deixar a entrada superior da ULA sempre recebendo o valor de um registrador (o rs1, claro, comum em ambos os formatos). A entrada inferior irá receber ou a constante estendida, se a instrução for do formato I, ou o registrador rs2, se a instrução for do formato R. Ambos a constante e o valor lido de rs2 entram em um multiplexador para que este escolha uma das duas.

Isso é feito de maneira simples: se a instrução for um `add s1,s2,s3`, o opcode 0110011_2 irá indicar o formato R e a porta lógica irá produzir um

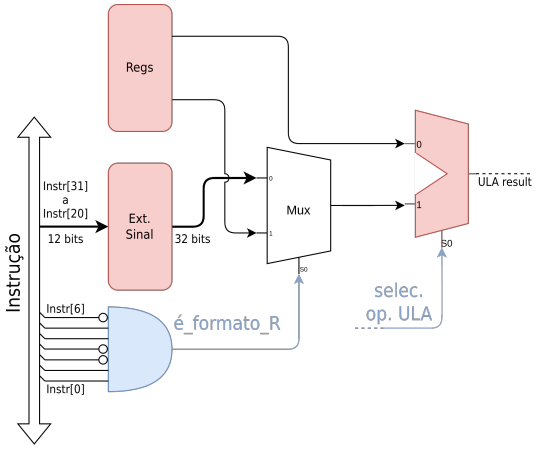

Figura 8: Controle do mux de seleção pelo formato R/I da instrução

sinal 1 à saída, cujo sinal está denominado como `é_formato_R`, selecionado o valor de s3 a ser passado para a ULA, o que se vê na figura 8.

Para a instrução `addi s1,s2,51`, o opcode 0010011_2 do formato I irá fazer a porta produzir um nível 0, identificando um formato diferente do R.[4] O mux irá portanto selecionar a entrada com a constante 51 estendida para 32 bits e irá repassar este valor à ULA, que finalmente vai proceder com a soma.

6 Condicionais

A maioria dos processadores faz as comparações por instruções da ULA, que alteram indicadores (as *flags,* que são posteriormente verificadas pela instrução de desvio propriamente dita). O RISC-V tem uma abordagem diferente: *a comparação em si* é feita pela mesma instrução que vai atualizar o endereço destino no PC. Observe que, neste documento, vamos analisar apenas o caso da instrução `beq` (*branch if equal*); as demais comparações funcionam de forma similar.

6.1 Cálculo do "delta"

O número de endereços a serem saltados pela instrução está incluído nos 32 bits do código binário dela, dados pelos campos imm1 e imm2 do formato B de instruções, como foi visto no laboratório 3 de assembly. [5] Portanto

[4] Aqui vamos nos ater a estes casos restritos. Para expandir, basta utilizar um multiplexador com mais entradas, cobrindo todos os casos, uma porta para identificar o formato I e uma combinação adequada destes sinais para acionar o mux.

[5] Uma tabela completa de instruções e formatos do core RV32I e das extensões padrão pode ser vista em https://github.com/jameslzhu/riscv-card/blob/master/riscv-card.pdf,

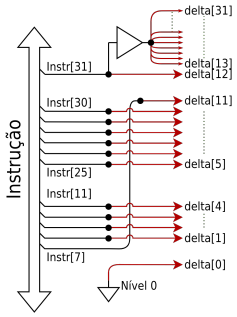

Figura 9: Produção da constante com a quantidade de endereços de memória para o salto condicional

precisamos obter os bits dos campos imm1 e imm2, reorganizá-los na posição final (pois estão embaralhados), incluir um zero no LSB à direita e finalmente estender o sinal da constante para que fique com 32 bits, de forma inversa ao explicado no laboratório.

O circuito é bastante simples, na verdade: basta manipular fios no circuito, como vemos na figura 9. Perceba apenas a indicação de um buffer (algo como uma "porta não inversora") para lidar com a alta corrente de fanout para o bit de sinal poder ser derivado para incluir 19 linhas adicionais na extensão de sinal.

Este número binário resultante será finalmente somado ao PC, realizando então o salto em si.

6.2 Seleção do Próximo Endereço

Para determinar igualdade entre dois operandos, a ULA pode fazer uma simples subtração: se o resultado da conta for zero, os números são iguais. Convenientemente, basta colocar na ULA uma porta lógica NOR entre todos os bits do resultado, que portanto vai indicar nível 0 se qualquer um dos bits acusar diferença. Vamos chamar este sinal de `é_igual`, já que ele indica que as duas entradas da ULA são idênticas.

Precisamos habilitar o salto apenas para a instrução `beq`, que possui o opcode 1100011_2 e o funct3 em 000_2. Faremos a decodificação de forma similar ao que fizemos para o `sw`, como na seção 5, gerando um sinal `instr_beq` que fica ativo (em 1) apenas para esta instrução e desativado (em 0) para as outras.

A instrução `beq` vai saltar quando os dois operandos forem iguais, portanto o PC deve ser escrito com o valor PC+delta_endereços quando ambos `é_igual==1` e `instr_beq==1`; caso contrário, devemos escrever PC+4 no PC. Isso nos dá um simples mux com os dois valores de escrita como en-

como indicado nas referências do Moodle.

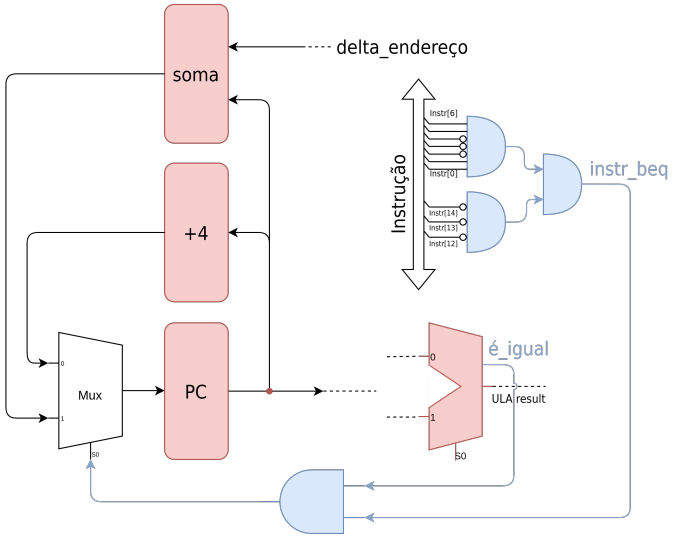

Figura 10: Seleção do endereço da instrução a ser executada após a atual (note que em azul estão componentes e sinais da Unidade de Controle)

tradas e uma AND entre `é_igual` e `instr_beq` funcionando como seleção, como podemos ver na figura 10.

No caso de incluirmos outras instruções de saltos, podemos gerar mais sinais pela ULA e combiná-los, dentro ou fora da ULA, e produzir um sinal `comparação_válida` ou algo similar.

7 Circuito Completo

Unindo todos os pedaços do circuito descritos anteriormente, temos o circuito completo para um microprocessador RISC-V ciclo único na figura 11. Note, em especial, que toda a decodificação está abstraída, contida na UC, mas mesmo assim o circuito está bastante completo e compreensível.

7.1 Caminhos de Dados e de Controle

Há uma convenção fundamental de arquitetura de computadores ilustrada intuitivamente em nosso circuito: o assim chamado caminho de dados está em preto e vermelho, com o caminho de controle em azul.

As setas em preto, da esquerda para a direita, mostram como os dados em si trafegam dentro do processador, iniciando com um endereço (PC), transformado em uma instrução (lida da ROM) a partir da qual se identifica os operandos (pelo banco de registradores ou constantes imediatas) que serão fornecidos à ULA ou à RAM para produzir o resultado. Os resultados estão marcados em vermelho e têm o sentido da direita para a esquerda.

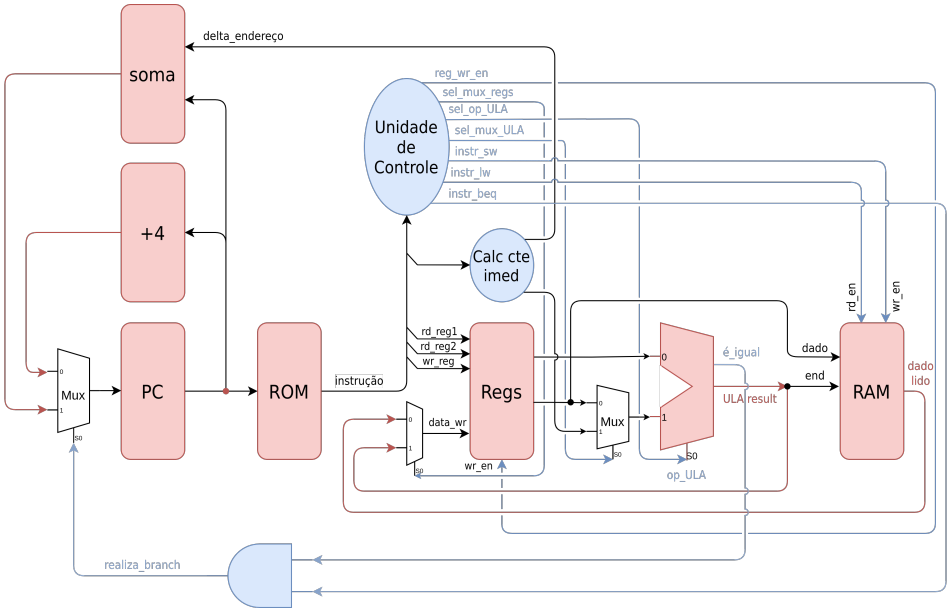

Figura 11: Microprocessador ciclo único completo

Já o caminho de controle é mais óbvio, tratando-se de todos os sinais gerados pela unidade de controle, ou seja, os sinais de enable para os blocos e os sinais de seleção para os muxes, todos identificados pela cor azul. Lembre que a unidade de controle é apenas combinacional, não possuindo nenhum estado ou flip-flop.

7.2 Detalhes Novos

O sinal `op_ULA` indica para a ULA qual operação deve ser realizada. Para os nossos propósitos, instruções de soma como add e addi produzem uma soma, e a instrução de condicional beq, que exige uma comparação, solicita uma subtração à ULA. Outras instruções, como sub, bne ou bge criam novos casos a tratar, mas construir este decodificador é trivial: é uma simples tabela com instruções do lado esquerdo e a conta a realizar do lado direito e até um mapa de Karnaugh pode resolver isso.

Também temos o sinal `reg_wr_en` (*write enable* dos registradores) gerado pela UC e conectado ao wr_en do Banco de Registradores, controlando assim quando há uma escrita em um registrador e quando não há. São poucas as instruções do core RISC-V que não escrevem um resultado no banco; as principais são desvios (que escrevem no PC) e sw (que escreve na RAM), com algumas outras secundárias que não veremos.

No nosso circuito, portanto, `reg_wr_en` só vai ficar em zero quando a instrução sendo executada for beq ou sw e vai ficar em 1 nos demais casos, o que nos dá a equação:

```
reg_wr_en <= NOT (instr_sw OR instr_beq)
```

7.3 RAM

Apenas as instruções `lw` e `sw` vão utilizar a RAM. A geração dos *enables* de leitura e escrita foram vistas em detalhes na seção 5, mas faltou ver o cálculo do endereço e a movimentação dos dados.

Endereços

O modo de endereçamento básico do RISC-V é o indexado, ou seja, o endereço a ser acessado na RAM é obtido pela soma de um ponteiro base com uma constante (um "delta" de deslocamento): um `lw s1,28(s2)` deverá ler o endereço de memória dado por s2+28 e escrever o dado lido em s1. De forma análoga, um `sw s3,8(s7)` vai escrever o valor de s3 no endereço dado por s7+8. Os modos direto e indireto são pseudoinstruções.

A soma do ponteiro com a constante é feita pela ULA da seguinte forma: o valor do ponteiro é identificado pelo campo rs1 da instrução, como sempre, que é direcionado ao banco de registradores, fazendo com que o valor do ponteiro seja apresentado nos pinos rd_reg1, ligados à entrada superior da ULA. Na entrada inferior devemos ter a constante mas, como já vimos, outras instruções usam o valor de um registrador nesta entrada (pense num `add s1,s2,s3`). Precisamos então de um mux para escolher entre esta constante e o valor do registrador, e o sinal `sel_mux_ULA` será usado para isso, como visto anteriormente.

No caso das instruções lw e sw, portanto, o sinal deverá estar valendo 1 para escolher a entrada da constante. Isto é consistente com nossa explicação anterior (o formato I exige que o sinal vá para 1), mas há a complicação adicional de sw ter o formato S, que também precisa de `sel_mux_ULA` em 1. Uma lógica simples dá conta disso, examinando o opcode e o funct3.

Também é preciso observar que o formato S e o formato I têm os bits da constante em lugares diferentes do código de máquina da instrução; assumimos que este "desembaralhamento" é realizado pelo bloco de *cálculo da constante imediata,* que também vai resolver a extensão de sinal.

Dados

O dado escrito por uma instrução `sw` deve vir de um registrador lido do banco; se observarmos o formato da instrução S no manual, veremos que o registrador é identificado pelo campo rs2 da instrução, como era de se esperar, chamado de rd_reg2 na figura 11. Portanto, este dado estará disponível na saída inferior do banco de registradores (identificada como data_r2 na figura 5) e irá até os pinos de dados a escrever na RAM.

Já o dado lido por uma instrução `lw` vai ser colocado na saída da RAM que está ligada na entrada de um mux (confira a figura 11). Este mux irá

selecionar qual dos dados será repassado ao banco de registradores para ser escrito, ou seja, é o resultado da operação. Nos casos em que não há escrita de resultado em um registrador, ele será ignorado como lixo, pois o wr_en do banco estará desabilitado – isso se vê em instruções como sw ou beq, por exemplo.

O pino de seleção deste mux tem, portanto, uma função simples: deve escolher a entrada 0, vinda da RAM, apenas quando a instrução for `lw`, escolhendo a entrada 1 em todas as outras instruções. Portanto, basta fazer uma decodificação similar à da seção 5 que identifica se a instrução é lw e gerar nível zero neste caso e apenas neste caso. Isto nos dá as equações:

```
instr_lw <= (instr[6..0]==0000011) AND (instr[14..12]==010)
sel_mux_regs <= NOT instr_lw
```

O sinal instr_lw indica com 1 se a instrução atual é lw ou com 0 no caso contrário. Como isto é o inverso do que desejamos para o mux, basta inserir uma porta NOT neste sinal e ligá-lo ao mux.

8 Introdução a Microprocessadores Multiciclo

Microprocessadores ciclo único são normalmente um artefato histórico; quase todos os processadores modernos usam pipelining (uma forma de paralelização que será vista mais à frente) pela simples razão que o desempenho é bem melhor e pode ser ainda mais otimizado com técnicas avançadas. Os multiciclo têm uma utilização bastante restrita, mas servem pedagogicamente como base para compreensão dos circuitos mais complexos.

8.1 Qual a Vantagem do Multiciclo sobre o Ciclo Único?

A resposta é muito simples: *desempenho* (por favor, não use o termo "velocidade," o processador não tem pernas, ele não corre). E por que fica mais rápido? ;-)

Todas as operações no ciclo único devem levar no máximo o período de um ciclo de clock para executar. Isso implica que se, por exemplo, tivermos um clock de 100 MHz, a propagação dos dados pelas portas lógicas até a estabilização de todos os níveis deve levar no máximo 10 ns, ou, dito de outra forma, *a operação mais lenta da ISA deve ser completamente realizada no tempo máximo de 10 ns.* Se quisermos colocar uma operação muito lenta (digamos a divisão de ponto flutuante), teremos que baixar este clock.

No multiciclo, temos uma solução trivial para este problema: vamos dividir todas as instruções em pedaços (são os estados) e a frequência máxima de clock será limitada pelo "pedaço" mais lento, que exige mais tempo de propagação. Se o clock ainda estiver baixo, podemos subdividir mais ainda a operação limitante.

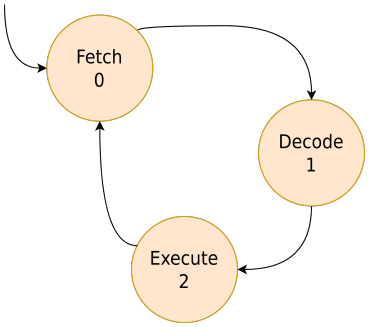

Figura 12: Máquina de estados de um multiciclo com 3 clocks

8.2 Usando Máquinas de Estado

Uma maneira simples de dividir a execução de uma instrução é quebrá-la em alguns estados padronizados. Um esquema simples e típico pode ser:

- **Fetch** (busca): lê a instrução da memória (e incrementa o PC);
- **Decode:** decodifica a instrução, identificando-a e gerando os sinais de controle necessários;
- **Execute:** realiza a operação em si e grava os resultados de acordo.

Uma máquina de estados correspondente pode ser vista na figura 12. O estado inicial no reset é fetch, e a cada clock temos uma transição, ciclicamente. Isso pode ser facilmente implementado como um simples contador. Podemos, é claro, fazer um processador com mais estados, ou com menos, dividindo as operações de forma diferente.

Também podemos fazer uma particularização para certas operações, a fim de otimizar a execução. Vamos criar um exemplo, com os estados vistos na figura 13: se incluirmos uma operação lenta (divisão, neste caso), isto causaria uma diminuição do clock, pois o estado Execute iria demorar mais por conta dela. Podemos então fazer o caminho padrão de 3 clocks para instruções normais, rápidas (estados 0=>1=>2), e fazer um caminho alternativo de 5 clocks que será percorrido apenas no caso da divisão (estados 0=>1=>4=>5=>6, na nossa solução).

Isso é conseguido fazendo-se uma transição condicional no estado 1, indo ou para o estado 2 ou para o 4, dependo da instrução ser ou não divisão. Desta forma, a maioria das instruções será executada rapidamente sem prejudicar o desempenho pela instrução lenta.

8.3 Dicas para Análise do Circuito

O circuito do livro-texto do Patterson faz uma gambiarra notável: ele aproveita pra usar a ULA mais de uma vez, calculando destinos de saltos, ou

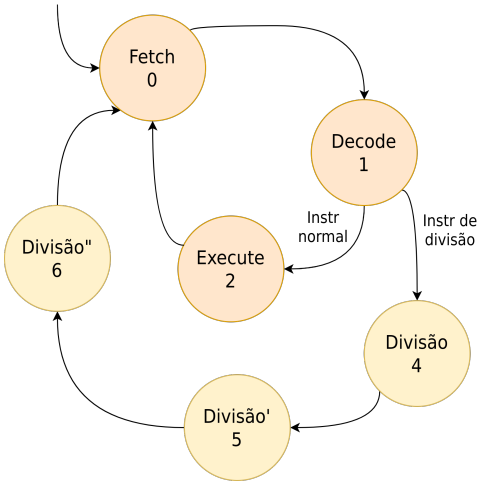

Figura 13: Máquina de estados de um multiciclo com dois caminhos de execução

seja, a ULA soma o PC com alguma coisa em um momento em que ela estaria desocupada. Pode não ser muito prático, afinal se colocarmos outros somadores para atualizar o PC gastaremos poucas portas adicionais, mas isso demonstra algumas possibilidades interessantes que o projetista tem ao seu alcance.

Portanto preste atenção ao que está acontecendo em cada momento, consultando a sequência dos sinais na máquina de estados.

Também é perceptível a adição de pequenos quadrados misteriosos (A, B, ALUOut e Memory data register); veja que na convenção gráfica utilizada, todo quadrado é uma memória (sejam flip-flops, registradores ou RAM). Estes blocos servem para segurar os dados de um clock para o clock seguinte, já que eles poderiam ser destruídos por lixo (ou por dados subsequentes) e são necessários no próximo estado da máquina.

Por fim, este processador tem arquitetura Harvard ou von Neumann? Por quê?