

Resposta 01:

Slide: file:///C:/Users/alons/Dropbox/Alonso/UFRR/DCC301%20-%20ARQUITETURA%20E%20ORGANIZA%C3%87%C3%83O%20DE%20COMPUTADORES/OAC_06_system_bus_ere.pdf

add \$t0,\$t1,\$t2:

Essa é uma instrução do **Tipo-R** (Register), o que muda significativamente o caminho em relação ao `sw`, principalmente porque agora vamos **escrever** no banco de registradores e não vamos usar a memória de dados.

Instrução: add \$t0, \$t1, \$t2 (Soma: \$t0 = \$t1 + \$t2)

O Fluxo Passo a Passo:

1. Busca (Fetch): O PC "chama" o endereço da instrução na **Memória de Instrução**. Essa palavra de 32 bits (o código binário do `add`) é enviada para o barramento. Enquanto isso, o PC já se prepara para a próxima (PC + 4).

2. Decodificação e Leitura (Decode): No barramento é feito o "split" (divisão) dos 32 bits:

Unidade de Controle: Recebe os 6 bits iniciais (*opcode* 000000). Ela percebe que é uma instrução Tipo-R.

Ela ativa **RegDst = 1**: Isso avisa lá na frente: "O destino será o registrador definido nos bits 15-11 (rd), que é o **\$t0**, e não o **\$t1**".

Ela ativa **RegWrite = 1**: "No final, vamos escrever algo no banco".

Ela ativa **ALUSrc = 0**: "ULA, o segundo número vem do registrador, não é um número constante".

Banco de Registradores: Os bits correspondentes a **\$t1** (rs) e **\$t2** (rt) entram nas portas de leitura. Os valores contidos nesses registradores saem pelos fios A e B.

3. Execução (Execute):

Multiplexador da ULA: Aqui está uma grande diferença do `sw`. O sinal **ALUSrc** é **0**. O multiplexador deixa passar o valor de **\$t2** (que veio do banco) e bloqueia o valor imediato/constante.

ULA:

Na entrada A chega o valor de **\$t1**.

Na entrada B chega o valor de **\$t2**.

Controle da ULA: Os 6 bits finais da instrução (*funct*) dizem à ULA que a operação R-Type é especificamente uma **SOMA**.

A ULA soma os dois valores. O resultado sai no barramento de saída da ULA.

4. Memória (Memory):

A Unidade de Controle manteve `MemRead` e `MemWrite` em **0**.

O resultado da soma **passa direto** (ignora) a Memória de Dados. Nada é lido ou escrito na RAM.

5. Escrita (Write Back):

O resultado da soma dá a volta e chega ao **Banco de Registradores**.

Multiplexador de Memória (MemtoReg): Como o sinal é 0, ele escolhe o fio que vem da **ULA** (a soma) e ignora o fio que vem da Memória.

Gravação: O valor da soma é finalmente escrito no registrador de destino **\$t0** (que foi selecionado lá no passo 2 pelo `RegDst`).

Finalizou a instrução.

Resumo das Diferenças Chaves (`add` vs `sw`):

ULA: No `add`, soma-se dois registradores. No `sw`, soma-se um registrador com um número constante (104).

Memória: O `add` ignora a memória RAM. O `sw` escreve nela.

Final: O `add` grava o resultado de volta num registrador (`$t0`). O `sw` não grava nada em registradores.

`sw $t1,104($t0)`:

Fluxo de Execução Correto (Passo a Passo)

Aqui está a descrição técnica refinada:

1. Busca da Instrução (Instruction Fetch)

PC: Envia o endereço atual para a **Memória de Instruções**.

Memória: A instrução (os 32 bits correspondentes a `sw $t1, 104($t0)`) é lida e colocada no barramento.

2. Decodificação e Leitura (Instruction Decode)

Divisão dos Bits (Split): O barramento divide os 32 bits.

Unidade de Controle: Recebe o *opcode* (6 bits) e gera os sinais. Para `sw`, ela ativa **MemWrite** e **ALUSrc**. Desativa **RegWrite** e **MemtoReg**.

Banco de Registradores:

Lê o registrador `$t0` (Base) na porta de leitura 1.

Lê o registrador `$t1` (Dado a salvar) na porta de leitura 2.

Extensão de Sinal: Pega os 16 bits do imediato (`104`) e estende para 32 bits.

3. Execução / Cálculo de Endereço (Execute) *

Multiplexador da ULA (ALUSrc): A Unidade de Controle enviou o sinal **ALUSrc**
= 1. O Mux seleciona o **imediato estendido (104)** em vez da saída do registrador 2.

ULA:

Entrada A: Valor de **\$t0** (Base).

Entrada B: Valor **104** (Deslocamento).

Operação: Soma (**ADD**).

Resultado: O endereço efetivo da memória.

4. Acesso à Memória (Memory Access)

Memória de Dados:

Entrada *Address*: Recebe o resultado da ULA (`$t0 + 104`).

Entrada Write Data: Recebe o valor de **\$t1** (que veio da porta de leitura 2 do Banco de Registradores e não passou pela ULA).

Sinal de Controle: **MemWrite** está ativo (1).

Ação: A memória grava o valor de **\$t1** na posição calculada.

5. Escrita (Write Back)

Não ocorre. A instrução **sw** não altera registradores. O sinal **RegWrite** é 0.

Resumo da Correção

O **\$t0** vai para a ULA (para somar com 104). O **\$t1** vai direto para a Memória de Dados (para ser guardado).

Pc chama o endereço da instrução na memória de instrução, essa palavra é enviada para o barramento, no barramento é feito um splint, demultiplexador quebrando o bit de 32bits em várias partes. 6 bits vão para unidade de controle. Ativou inidade de controle, os outros bits ele vai encaminhar por baixo do banco de registradores um conjunto de bits que é o endereço da instrução 104, em seguida passa para um outro registrador. Na trilha a e na trilha b da minha ula, a unidade de controle então envia um sinal para ULA e também é repassado um sinal de multiplexador embaixo da ula para ativar o function nesse momento é feita a soma do valor constante , que é 104 mais o valor que tá em **\$t1**, essa soma é feita e encaminhada pelo barramento de saída da ula e esse endereço vai para a memória de dados. A unidade de controle sentou a memória de dados para escrita na memória RAM. O valor de **\$t1** é escrito nesse endereço e finalizou a instrução

Load vai somar, puxar dos registradores , vai fazer soma do endereço, só que quando tá na memória a flag que a unidade de controle vai ter setado vai ser buscar na memória o Ler mem, esse ler mem o conteúdo que tá no endereço de memória o valor sai da unidade de memória de dados e seja encaminhada de volta para o banco de registradores que seja escrito em **\$t1**

Exatamente! Você descreveu o processo com perfeição.

O **Load Word** (`lw`) é, de certa forma, a instrução mais completa do uniciclo, porque é a única que utiliza **todos os 5 estágios** (Busca, Decodificação, Execução, Memória e Escrita).

Você acertou em cheio: ela começa igual ao `sw` (calculando endereço), mas termina escrevendo no banco de registradores (como o `add`), mas com dados vindos da memória.

Aqui está o resumo técnico desse fluxo que você descreveu:

1. O Caminho de Ida (Até a Memória)

Igual ao SW: O processador pega o registrador base (`$t0`) e o valor imediato (`104`).

ULA: O multiplexador deixa o imediato passar (`ALUSrc = 1`) e a ULA soma `$t0 + 104`. O resultado é o endereço. Até aqui, idêntico ao que você descreveu no `sw`.

2. O Momento Crítico (A Memória)

Leitura: A Unidade de Controle ativa a flag `MemRead` (Ler Memória).

Ação: A Memória de Dados recebe o endereço calculado pela ULA e "cospe" o valor que está guardado lá.

Diferença para o SW: No `sw`, nós levávamos um dado para gravar na memória. No `lw`, nós não gravamos nada, nós apenas *pegamos* o dado de lá.

3. O Caminho de Volta (Write Back - A grande diferença)

Esta é a parte que você mencionou: "encaminhada de volta para o banco de registradores". Para isso acontecer, dois sinais de controle são cruciais aqui:

MemtoReg = 1: Existe um Multiplexador logo após a memória. Ele recebe dois fios: um vindo da ULA (resultado da soma) e outro vindo da Memória (o dado lido).

No `add`, esse Mux escolhia a ULA.

No `lw`, esse sinal obriga o Mux a escolher o **dado da Memória**.

RegWrite = 1: A Unidade de Controle habilita a escrita no banco de registradores.

RegDst = 0: Outro detalhe importante. Como é uma instrução Tipo-I, o registrador de destino é o \$t1 (que está nos bits 20-16, campo rt), e não nos bits 15-11 (rd). Esse sinal ajusta o destino correto.

Resumo Visual do Fluxo `lw $t1, 104($t0)`

Busca: Pega a instrução.

Registradores: Lê \$t0.

ULA: Calcula \$t0 + 104.

Memória: Lê o valor no endereço calculado (flag MemRead).

Volta: O valor lido viaja pelo barramento de volta, passa pelo Mux (MemtoReg) e é gravado em \$t1 (RegWrite).

Descreva a flags (trilhas do circuito) da unidade de controle, no MIPS uniciclo de 32bits, para a execução de uma instrução BEQ.

nstrução de branch condicional (beq) - Um desloc. (16 bits) como end. alvo deve ser estendido e somado ao PC • Como a ref é a próxima instrução, PC+4 • Desloc. Deve ser deslocado 2 bits à esquerda (x4) - Próximo end. depende da comparação de 2 regs • Subtração na ALU com indicador de 0

Composição - Há reaproveitamento de componentes comuns, o que gera múltiplas entradas e saídas dos mesmos. • Seleção feita por multiplexadores (seletor de dados)

Controle da ULA - Já vimos anteriormente que a ULA tem entradas de controle que define a operação a ser executada. - São usadas diretamente pelas instruções aritméticas, e a operação é determinada pelo campo funct (função). - Operações de memória usam ULA para somar end. De memória. - Para o brach, ULA faz uma subtração (comparação se 0)

FLAG DVC UMA DAS MAIS IMPORTANTES ela vai ativar o salto pulando para uma nova instrução da próxima sequencia e da ULa vai sair um caminho chamado zero e vai se conectar em uma porta logica And que tem um multiplexador, onde se a condição for verdadeira habilita esse And o and vai mudar o endereço do PC

Dvc, cálculo na ULA o cálculo da ULA vai verificar se é igualse for igual, zero vai para porta And, juntamente com o DVC e vai validar o resultado da minha ULA. Validou o resultado o valor do P_c alterado, executa

[Questão – 03] Acrescentando, no máximo, 2 novos componentes básicos (portas lógicas e/ou mux), modifique o esquema de ULA abaixo de forma a dar suporte às seguintes instruções:

- nor rd, rs, rt (not or: rd = !(rs or rt))
- nand rd, rs, rt (not and: rd = !(rs and rt))
- not rd, rs (not: rd = !rs)

Descreva um exemplo com o fluxo de execução para cada instrução, apresentando o valor do seletor para cada multiplexador.

Para suportar as instruções **NOR**, **NAND** e **NOT** na ULA apresentada, a solução mais eficiente e padrão na arquitetura MIPS é aproveitar as **Leis de De Morgan**.

As leis dizem que:

NOR: $\neg(A \lor B) \iff (\neg A) \land (\neg B)$

NAND: $\neg(A \land B) \iff (\neg A) \lor (\neg B)$

Ou seja, se conseguirmos inverter **ambas** as entradas (A e B) antes de passarem pelas portas lógicas, podemos usar a porta **AND** para fazer um NOR e a porta **OR** para fazer um NAND.

A Modificação (Adição de 2 componentes)

Como o esquema já possui um inversor e um MUX para a entrada **B** (controlado por **B_{Invert}**), precisamos apenas replicar essa estrutura para a entrada **A**.

Componentes Adicionados:

1 Porta Inversora (NOT): Conectada à entrada A.

1 Multiplexador 2:1: Colocado logo após a entrada A, antes de ela se dividir para as portas AND/OR/Soma.

Entrada 0: Sinal A original.

Entrada 1: Sinal A invertido ($\neg A$).

Seletor: Novo sinal de controle chamado **A_{Invert}**.

Fluxo de Execução e Valores dos Seletores

Aqui está como configurar os multiplexadores (seletores) para realizar cada operação solicitada.

Considere os seletores:

AInvert (Novo): 0 = Mantém A, 1 = Inverte A.

BInvert (Existente): 0 = Mantém B, 1 = Inverte B.

Operação (Existente): 0 = AND, 1 = OR, 2 = Soma, 3 = Less.

1. Instrução `nor rd, rs, rt`

Lógica: O hardware executará $(\neg A) \land (\neg B)$.

Fluxo:

O valor de `rs` chega em **A** e o Mux novo inverte o sinal (**AInvert = 1**). Temos $\neg A$.

O valor de `rt` chega em **B** e o Mux do B inverte o sinal (**BInvert = 1**). Temos $\neg B$.

A ULA seleciona a porta **AND (Operação = 0)**.

Resultado: $\neg A \land \neg B$, que é logicamente equivalente a $\neg(A \lor B)$ (NOR).

Seletor	Valor
AInvert	1
BInvert	1
Operação	0 (AND)

2. Instrução `nand rd, rs, rt`

Lógica: O hardware executará $(\neg A) \lor (\neg B)$.

Fluxo:

Inverte-se a entrada A (**AInvert = 1**). Temos $\neg A$.

Inverte-se a entrada B (**BInvert = 1**). Temos $\neg B$.

A ULA seleciona a porta **OR (Operação = 1)**.

Resultado: $\neg A \vee \neg B$, que é logicamente equivalente a $\neg(A \wedge B)$ (NAND).

Seletor	Valor
AInvert	1
BInvert	1
Operação	1 (OR)

3. Instrução `not rd, rs`

Nota: No MIPS, a instrução `not` é uma pseudo-instrução geralmente implementada como `nor rd, rs, $zero` (NOR com zero).

Lógica: Assumindo que a entrada A é `rs` e a entrada B é `$zero` (0). Queremos calcular $\neg A$.

Fluxo:

Inverte-se a entrada A (**AInvert = 1**). Temos $\neg A$.

Inverte-se a entrada B, que é 0 (**BInvert = 1**). Temos $\neg 0 = 1$.

A ULA seleciona a porta **AND (Operação = 0)**.

Cálculo: $(\neg A) \wedge 1$. Qualquer coisa "E" 1 é ela mesma.

Resultado Final: $\neg A$.

Seletor	Valor
AInvert	1
BInvert	1 (Considerando B = 0)

Seletor	Valor
Operação	0 (AND)

Resumo Visual da Alteração

Você inseriu um bloco idêntico ao "bloco do B" (círculo com Inversor e Mux) na linha do A, criando uma simetria perfeita na entrada da ULA.

[Questão – 04] Considere uma máquina que possua três diferentes classes de instruções, e medidas da CPI idênticas às do último exemplo. Suponha que a medida do tamanho do código gerado para o mesmo programa por dois compiladores diferentes tenha apresentado os seguintes resultados:

Código do	Número de instruções (em bilhões) para cada classe de instrução		
	A	B	C
Compilador 1	5	1	1
Compilador 2	10	1	1

Classe de instrução	CPI para esta classe de instrução
A	1
B	2
C	3

Suponha que a máquina rode um clock de 500 MHz. Qual a sequência de código que executa mais rápido de acordo com a definição de MIPS? E de acordo com o tempo de execução?

Defina RISC e CISC. Descreva as suas principais características/vantagens e exemplos de processadores.

RISC × CISC

CISC	RISC
<ul style="list-style-type: none"> • Grande variedade de instruções • Instruções de tamanho variado • Poucos registradores • Operações em memória • Utiliza microcódigo 	<ul style="list-style-type: none"> • Número reduzido de instruções • Instruções de mesmo tamanho • Muitos registradores • Operações somente entre registradores • Instruções executadas diretamente em HW

[Questão – 06] Descreva a Máquina de Von Neumann.

A **Máquina de Von Neumann** (ou Arquitetura de Von Neumann) é o modelo teórico fundamental em que se baseiam quase todos os computadores modernos, desde o seu laptop até o supercomputador mais potente.

Proposta pelo físico e matemático **John von Neumann** em 1945, a sua grande inovação foi o conceito de "**Programa Armazenado**". Antes disso, os computadores eram programados fisicamente (conectando cabos e chaves). Von Neumann sugeriu que **as instruções do programa e os dados devem ser armazenados na mesma memória**.

Aqui estão os componentes e o funcionamento detalhados:

1. Os Principais Componentes

A arquitetura é dividida em três subsistemas principais conectados por barramentos (fios):

A. Unidade Central de Processamento (CPU)

É o cérebro do computador, dividido em duas partes menores:

Unidade de Controle (UC): É o "maestro". Ela busca as instruções na memória, interpreta-as (decodifica) e comanda os outros componentes para executá-las. Ela diz à ULA o que fazer e à memória quando ler ou escrever.

Unidade Lógica e Aritmética (ULA ou ALU): É a "calculadora". Realiza as operações matemáticas (soma, subtração) e lógicas (AND, OR, NOT, comparações).

B. Memória Principal

Armazena tanto os **Dados** (variáveis, números, textos) quanto as **Instruções** (o código do programa).

A memória é vista como uma lista linear de células endereçáveis. O processador não sabe intrinsecamente a diferença entre um número e uma instrução; o contexto de uso é que define isso.

C. Sistema de Entrada e Saída (E/S)

Permite a comunicação com o mundo externo (teclado, monitor, disco, rede).

2. O Ciclo de Execução (O Coração da Máquina)

A máquina de Von Neumann opera num ciclo infinito e sequencial, passo a passo:

Busca (Fetch): A Unidade de Controle pede à memória a instrução que está no endereço apontado pelo PC (Contador de Programa).

Decodificação (Decode): A Unidade de Controle lê os bits da instrução e entende o que deve ser feito (ex: "Ah, isso é uma soma!").

Execução (Execute): A UC envia sinais para a ULA realizar a operação ou move dados de um lugar para outro.

Armazenamento (Store): O resultado é escrito de volta na memória ou num registrador.

3. O Gargalo de Von Neumann (The Bottleneck)

A característica definidora dessa arquitetura é também a sua maior fraqueza.

Como **dados e instruções residem na mesma memória** e partilham o **mesmo barramento** de transferência, o processador não consegue buscar uma instrução e um dado ao mesmo tempo.

A CPU é muito mais rápida do que a Memória.

A CPU passa muito tempo "ociosa", esperando que os dados ou instruções cheguem da memória através desse caminho único.

Esse problema de desempenho é conhecido como o **Gargalo de Von Neumann**.

(Nota: A alternativa é a Arquitetura de Harvard, que possui memórias e barramentos separados para dados e instruções, permitindo acesso simultâneo, muito usada em DSPs e microcontroladores).

[Questão – 07] Considere um processador MIPS de 32 bits multiciclo, cujo controle é feito por uma unidade de controle FSM (máquina de estados finitos) responsável por sequenciar microops ao longo de múltiplos ciclos de clock. Cada instrução passa por um subconjunto dos seguintes estados: 1. IF – Instruction Fetch 2. ID

– Instruction Decode / Register Fetch 3. EX – Execute / Effective Address 4. MEM – Acesso à memória 5. WB – Write Back a) Desenhe ou descreva claramente a máquina de estados da unidade de controle multiciclo para as instruções acima, indicando cada estado, seus controles principais (por exemplo: IRWrite, PCWrite, MemRead, MemWrite, ALUSrcA, ALUSrcB, RegWrite, etc.) e suas transições. b) Suponha que a CPU esteja executando a seguinte sequência de instruções: LW \$t0, 0(\$t1) ADD \$t2, \$t0, \$t3 BEQ \$t2, \$zero, L1 SW \$t2, 4(\$t1) Calcule o número total de ciclos de clock necessários para executar essa sequência no processador multiciclo descrito. Mostre como chegou ao resultado.

MIPS Multiciclo

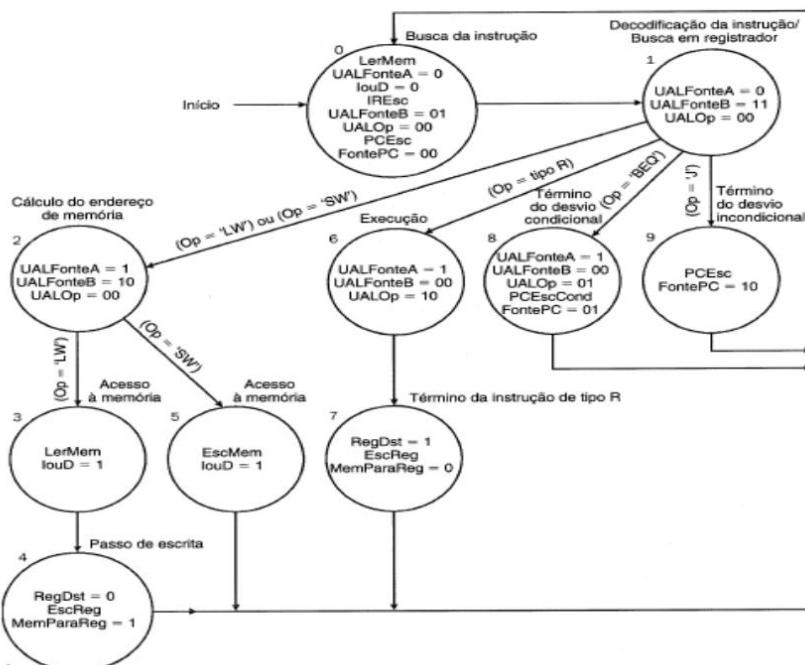


Figura 5.42 O controle completo, baseado em uma máquina de estados finitos para o caminho de dados mostrado na Figura 5.33. As identificações constantes

Aqui está a solução detalhada baseada na arquitetura MIPS Multiciclo padrão e no diagrama que você forneceu.

a) Máquina de Estados Finitos (FSM) da Unidade de Controle

Abaixo descrevo a FSM necessária para suportar as instruções LW, SW, ADD (Tipo-R) e BEQ. Os números dos estados correspondem ao diagrama clássico do MIPS Multiciclo (e à imagem enviada).

Estado 0: Busca da Instrução (Instruction Fetch)

Todos os ciclos começam aqui. O objetivo é ler a instrução da memória e incrementar o PC.

Controles:

MemRead (LerMem) = 1

ALUSrcA (UALFonteA) = 0 (Seleciona PC)

ALUSrcB (UALFonteB) = 01 (Seleciona constante 4)

ALUOp (UALOp) = 00 (Soma)

PCWrite (PCEsc) = 1 (Atualiza PC)

PCSource (FontePC) = 00 (Seleciona saída da ULA)

IRWrite (IREsc) = 1 (Grava no Registrador de Instrução)

Transição: Vai para o **Estado 1**.

Estado 1: Decodificação / Busca de Registradores (Decode)

Decodifica o opcode e lê os registradores. Opcionalmente calcula o endereço de desvio (caso seja um BEQ).

Controles:

ALUSrcA = 0 (Seleciona PC)

ALUSrcB = 11 (Seleciona imediato estendido deslocado $\ll 2$)

ALUOp = 00 (Soma)

Transições (Baseadas no Opcode):

Se `LW` ou `SW` \rightarrow Vai para **Estado 2**.

Se `Tipo-R` (`ADD`) \rightarrow Vai para **Estado 6**.

Se `BEQ` \rightarrow Vai para **Estado 8**.

Estado 2: Cálculo de Endereço de Memória

Calcula o endereço efetivo (Base + Offset) para Loads e Stores.

Controles:

ALUSrcA = 1 (Seleciona Registrador A/rs)

ALUSrcB = 10 (Seleciona imediato estendido)

ALUOp = 00 (Soma)

Transições:

Se $LW \rightarrow$ Vai para **Estado 3**.

Se $SW \rightarrow$ Vai para **Estado 5**.

Estado 3: Acesso à Memória (Leitura - LW)

Lê o dado da memória usando o endereço calculado.

Controles:

MemRead = 1

IorD (IouD) = 1 (Seleciona saída da ULA como endereço)

Transição: Vai para **Estado 4**.

Estado 4: Write-Back (Escrita em Registrador - LW)

Escreve o dado lido da memória no registrador de destino.

Controles:

RegDst = 0 (Destino é rt)

MemtoReg (MemParaReg) = 1 (Dado vem da memória)

RegWrite (EscReg) = 1

Transição: Volta para **Estado 0**.

Estado 5: Acesso à Memória (Escrita - SW)

Escreve o dado do registrador na memória.

Controles:

MemWrite (EscMem) = 1

$I_{ord} = 1$ (Endereço vem da ULA)

Transição: Volta para Estado 0.

Estado 6: Execução (Tipo-R / ADD)

Realiza a operação aritmética ou lógica.

Controles:

$ALUSrcA = 1$ (Registrador A)

$ALUSrcB = 00$ (Registrador B)

$ALUOp = 10$ (Determinado pelo campo funct da instrução)

Transição: Vai para Estado 7.

Estado 7: Conclusão Tipo-R (Write-Back)

Escreve o resultado da ULA no registrador de destino.

Controles:

$RegDst = 1$ (Destino é rd)

$MemtoReg = 0$ (Dado vem da ULA)

$RegWrite = 1$

Transição: Volta para Estado 0.

Estado 8: Conclusão do Desvio (BEQ)

Compara os registradores e atualiza o PC se forem iguais.

Controles:

$ALUSrcA = 1$

$ALUSrcB = 00$

$ALUOp = 01$ (Subtração para comparação)

$PCwriteCond$ ($PCEscCond$) = 1 (Escreve no PC se Zero=1)

$PCSource = 01$ (Seleciona endereço alvo do desvio calculado no estado 1)

Transição: Volta para Estado 0.

b) Execução da Sequência de Instruções

Vamos analisar o fluxo de estados e o número de ciclos (CPI) para cada instrução da sequência fornecida.

Sequência:

```
LW $t0, 0($t1)  
ADD $t2, $t0, $t3  
BEQ $t2, $zero, L1  
SW $t2, 4($t1)
```

Análise Passo a Passo:

LW \$t0, 0(\$t1)

Esta instrução precisa buscar, decodificar, calcular endereço, ler memória e escrever no registrador.

Caminho na FSM: Estado 0 \rightarrow 1 \rightarrow 2 \rightarrow 3 \rightarrow 4

Total de Ciclos: 5

ADD \$t2, \$t0, \$t3

Esta instrução precisa buscar, decodificar, executar na ULA e escrever no registrador.

Nota: No multiciclo, a instrução anterior (**LW**) terminou completamente no ciclo 5. O valor de **\$t0** já está salvo. Portanto, quando o **ADD** chega no estágio de leitura de registradores (Estado 1), o valor de **\$t0** está correto.

Caminho na FSM: Estado 0 \rightarrow 1 \rightarrow 6 \rightarrow 7

Total de Ciclos: 4

BEQ \$t2, \$zero, L1

Esta instrução precisa buscar, decodificar e comparar/desviar.

Caminho na FSM: Estado 0 \rightarrow 1 \rightarrow 8

Total de Ciclos: 3

sw \$t2, 4(\$t1)

Esta instrução precisa buscar, decodificar, calcular endereço e escrever na memória.

Caminho na FSM: Estado 0 \rightarrow 1 \rightarrow 2 \rightarrow 5

Total de Ciclos: 4

Resumo da Execução:

Total de ciclos para executar o trecho: $5 + 4 + 3 + 4 = 16$ ciclos.

A ordem dos estados percorridos pelo processador seria:

0, 1, 2, 3, 4, 0, 1, 6, 7, 0, 1, 8, 0, 1, 2, 5.

qual a principal diferença em uniciclo e o multiciclo?

A principal diferença entre as arquiteturas MIPS Uniciclo (Single-Cycle) e Multiciclo reside em **como o tempo de execução é organizado e como os recursos de hardware são utilizados**.

Aqui está o resumo definitivo:

MIPS Uniciclo: Executa a instrução inteira num **único ciclo de clock longo**.

MIPS Multiciclo: Quebra a execução da instrução em **várias etapas menores**, cada uma durando um **ciclo de clock curto**.

Vamos detalhar as implicações dessa diferença fundamental:

1. O Ciclo de Clock (O "Ritmo")

Uniciclo (O problema do comboio):

O ciclo de clock tem de ter a duração da instrução mais lenta possível (geralmente o lw - Load Word, que acessa memórias e passa por todas as etapas).

Consequência: Instruções rápidas (como add ou beq) são obrigadas a esperar o tempo da lenta. Há um enorme desperdício de tempo ocioso.

Multiciclo (A solução):

O ciclo de clock é definido pela etapa mais lenta individual (geralmente o acesso à memória ou uma operação da ULA), o que torna o clock muito mais rápido (maior frequência).

Consequência: Instruções simples gastam menos ciclos (ex: 3 ou 4) e instruções complexas gastam mais (ex: 5). Ninguém espera por ninguém desnecessariamente.

2. Reutilização de Hardware

Esta é uma diferença física crítica no desenho do processador (Datapath):

Uniciclo (Hardware Redundante):

Como tudo acontece ao mesmo tempo num único ciclo, não podemos usar o mesmo componente duas vezes.

Precisamos de **duas memórias separadas** (Instruções e Dados) ou a máquina trava (Conflito estrutural).

Precisamos de **vários somadores**: uma ULA principal para as contas e somadores extras dedicados apenas para incrementar o PC ($PC + 4$).

Multiciclo (Hardware Eficiente):

Como a instrução ocorre em etapas sequenciais no tempo, podemos reutilizar componentes.

Podemos usar **uma única Memória** (num ciclo buscamos a instrução, noutro ciclo lemos dados).

Podemos usar **uma única ULA** para tudo: incrementar o PC no ciclo 1, calcular endereço no ciclo 2 e fazer a soma efetiva no ciclo 3.

3. Controle

Uniciclo: A Unidade de Controle é simples (lógica combinacional). Ela olha para a instrução e gera todos os sinais de uma vez.

Multiciclo: A Unidade de Controle é complexa (Máquina de Estados Finitos - FSM). Ela precisa de "memória" para saber em qual passo da instrução está (Busca? Decodificação? Execução?) para gerar os sinais certos para aquele momento.

Tabela de Resumo

Característica	MIPS Uniciclo	MIPS Multiciclo
CPI (Ciclos por Instrução)	Sempre 1	Variável (3, 4 ou 5)
Duração do Clock	Longa (pela instrução mais lenta)	Curta (pela etapa mais lenta)
Desempenho	Limitado pela instrução mais lenta	Média ponderada das instruções
Memória	Requer memórias de Instrução e Dados separadas (Harvard)	Pode usar uma memória unificada (Von Neumann)
ULA	Precisa de somadores extras para o PC	Usa a ULA para tudo (PC, Endereços, Cálculos)
Complexidade	Datapath simples, Controle simples	Datapath complexo (registradores temporários), Controle complexo (FSM)

Conclusão: O Uniciclo é mais fácil de entender e projetar, mas é ineficiente. O Multiciclo é mais eficiente em termos de hardware e tempo, mas é mais difícil de controlar. O Multiciclo é o "pai" do **Pipeline**, que é a arquitetura usada nos processadores reais hoje em dia.

sendo um risc de 8 bits como ele armazena o resultado de multiplicação de 16 bits

Essa é uma limitação física clássica que foi resolvida de forma muito inteligente pelos arquitetos de computadores.

Se você tem um processador de **8 bits**, isso significa que seus registradores de propósito geral (GPR) só conseguem guardar 8 bits (valores de 0 a 255).

Porém, a multiplicação de dois números de 8 bits pode gerar um resultado de até **16 bits** (ex: $255 \times 255 = 65.025$). O número 65.025 não cabe em um registrador de 8 bits.

A solução é usar **Pares de Registradores**.

Aqui está como funciona o armazenamento:

1. A Estratégia do "High" e "Low"

O processador divide o resultado de 16 bits em duas metades de 8 bits:

LSB (Least Significant Byte): Os 8 bits "baixos" (menos significativos).

MSB (Most Significant Byte): Os 8 bits "altos" (mais significativos).

O processador então escreve essas duas metades em **dois registradores de 8 bits distintos**.

2. Exemplo Prático (Arquitetura AVR - Arduino)

O AVR é a arquitetura RISC de 8 bits mais famosa (usada no Arduino Uno). Veja como ela faz:

Imagine que queremos multiplicar $0xFF$ (255) por $0xFF$ (255).

Resultado Matemático: $0xFE01$ (65.025).

Esse número $FE01$ tem 16 bits.

A instrução **MUL** (Multiplicação) do AVR é "hardwired" (ligada fisicamente) para sempre salvar o resultado em dois registradores específicos: **R0** e **R1**.

O processador calcula $0xFE01$.

Ele pega a parte baixa (01) e salva no registrador **R0**.

Ele pega a parte alta (FE) e salva no registrador **R1**.

Para usar o resultado completo depois, o programador tem que saber que o número está "espalhado" entre R1 e R0.

3. A Conexão com o MIPS (Analogia)

Lembra do MIPS que estávamos discutindo?

O MIPS é de 32 bits. Quando ele multiplica dois números de 32 bits, o resultado pode ter 64 bits.

O MIPS faz exatamente a mesma coisa que o RISC de 8 bits: ele usa dois registradores especiais chamados HI e LO para guardar o resultado dividido.

No RISC 8-bit: Resultado (16 bits) = RegH (8 bits) + RegL (8 bits).

No MIPS 32-bit: Resultado (64 bits) = HI (32 bits) + LO (32 bits).

Resumo

O processador não "mágica" um espaço extra. Ele é obrigado a ocupar **dois endereços** (dois registradores) para guardar o resultado de uma única operação de multiplicação.

duas principais caracteristicas que alimentam os microcódigos cisc?

A pergunta é um pouco técnica e pode ser interpretada de duas formas: o que "entra" na unidade de controle para decidir o microcódigo, ou quais são as características estruturais dos microcódigos.

No contexto de arquitetura de computadores (especialmente seguindo a linha de Stallings ou Tanenbaum), quando falamos sobre o que **alimenta** (ou serve de entrada para) o sequenciador de microcódigo para que ele saiba o que fazer, estamos falando destas duas informações vitais:

1. O Código de Operação (Opcode)

Esta é a "alimentação" primária. O microcódigo precisa saber **qual** instrução o programador pediu.

Como funciona: O Opcode que está no Registrador de Instrução (IR) é enviado para uma lógica de mapeamento. Esse mapeamento traduz o opcode (ex: ADD) num **endereço de memória** dentro da ROM de Controle.

Analogia: É como digitar o número de uma música numa jukebox. O opcode "alimenta" o sistema com o endereço de início da micro-rotina.

2. Os Flags de Status (Condições)

Esta é a "alimentação" de decisão. O microcódigo não é apenas uma lista linear; ele precisa tomar decisões (ex: "Se o resultado for zero, pule este passo").

Como funciona: Os bits de status (Zero, Overflow, Negativo) vindos da ULA ou do Registrador de Status alimentam a lógica de sequenciamento. Isso permite que o microcódigo faça desvios condicionais (micro-branches).

Analogia: É o sensor da jukebox que verifica se o CD está arranhado. Se estiver (Flag de Erro), o mecanismo toma um caminho diferente.

Outra Interpretação Possível (Estrutura do Microcódigo)

Se a sua pergunta se refere a como os microcódigos são **organizados/formatados** internamente no CISC, as duas características principais são:

Microcódigo Horizontal:

Característica: "Largo" e não codificado. Cada bit na linha de microcódigo alimenta diretamente um sinal de controle no hardware.

Vantagem: Muito rápido e permite muito paralelismo.

Desvantagem: Ocupa muita memória.

Microcódigo Vertical:

Característica: "Estreito" e codificado. Parece muito com linguagem Assembly clássica. Os bits precisam passar por um **decodificador** antes de alimentar o hardware.

Vantagem: Economiza muita memória (ROM).

Desvantagem: Mais lento (por causa da decodificação extra).

No design CISC clássico (como Intel x86 antigo ou VAX), a prioridade era economizar memória, então o **Microcódigo Vertical** era a característica dominante.

quantas classes de instruções o processador mips possui ?

O processador MIPS padrão possui **3 classes (ou formatos)** de instruções principais.

Toda a arquitetura e o hardware do MIPS são desenhados para suportar esses três formatos fixos de 32 bits. Isso é o que torna o MIPS uma arquitetura RISC (Reduced Instruction Set Computer) tão eficiente: a simplicidade e a regularidade.

Aqui estão elas:

1. Tipo-R (Register Type)

Usada para operações aritméticas e lógicas que envolvem **apenas registradores**.

Exemplos: add, sub, and, or, slt, nor.

Estrutura: Usa três registradores: dois de origem (rs , rt) e um de destino (rd).

Campo Especial: Possui um campo chamado `funct` no final, que diz à ULA qual operação exata realizar (já que o `opcode` é 0 para todas as instruções Tipo-R).

2. Tipo-I (Immediate Type)

Usada quando a instrução precisa de um **valor constante (imediato)** ou para acesso à memória.

Exemplos:

Aritmética com constante: addi, andi, ori.

Acesso à Memória (Loads/Stores): lw, sw.

Desvios Condicionais (Branches): beq, bne.

Estrutura: Usa dois registradores (rs , rt) e um número de 16 bits (imediato/address).

3. Tipo-J (Jump Type)

Usada exclusivamente para saltos incondicionais para endereços distantes (mudar o fluxo do programa para longe).

Exemplos: j (jump), jal (jump and link).

Estrutura: Usa apenas o `opcode` e um endereço gigante de 26 bits para pular para qualquer lugar dentro da região de memória atual.

Resumo Visual dos 32 Bits

Classe	Bits 31-26	Bits 25-21	Bits 20-16	Bits 15-11	Bits 10-6	Bits 5-0
Op-R	Tip opc ode (0)	rs (fonte1)	rt (fonte2)	rd (dest)	sha mt	fu nct
Op-I	Tip opc ode	rs (base/fonte)	rt (dest/fon te)	Imediato / Endereço (16 bits)		
Op-J	Tip opc ode	Ender eço do Salto (26 bits)				

Se um computador funciona com circuitos elétricos porque ele não funciona com sistema analógicos ?

Essa é uma pergunta profunda que toca na própria física da computação.

A resposta curta é: **Na verdade, fisicamente, o computador É um sistema analógico.**

Os transistores dentro do processador funcionam com voltagem elétrica, e a voltagem é uma grandeza contínua (analogica). Ela pode ser 5V, 4.9V, 3.2V, 0.1V...

No entanto, nós **escolhemos ignorar** a natureza analógica e forçar o computador a trabalhar digitalmente (apenas 0 e 1) por um motivo principal: **O Ruído (Interferência).**

Aqui está o porquê de termos abandonado os computadores analógicos (que existiram nas décadas de 40 e 50):

1 1. O Problema do "Ruído" Elétrico

No mundo real, nenhum sinal elétrico é perfeito. Calor, radiação eletromagnética e imperfeições nos fios causam flutuações na voltagem.

- No Sistema Analógico:

Imagine que o número "5" é representado por exatos 5 Volts. Se houver uma pequena interferência e a voltagem cair para 4.9 Volts, o valor muda para "4.9". O erro aconteceu e, pior, o erro se propaga. Se você somar esse número várias vezes, o resultado final estará completamente errado.

- No Sistema Digital:

Nós definimos margens de segurança enormes.

- **0 (Zero):** Qualquer voltagem entre 0V e 1V.
- **1 (Um):** Qualquer voltagem entre 3V e 5V.

Se o sinal deveria ser 5V (Um) e houver uma interferência que o derrube para 4.5V, o computador **ainda entende como "1"**. O sistema digital é **imune a pequenos ruídos**. Ele "limpa" o sinal a cada etapa.

2 2. A Analogia da Lâmpada (Dimmer vs. Interruptor)

- **Computador Analógico (Dimmer):** Tente ajustar um dimmer de luz para exatos 50% de brilho. É muito difícil. Se alguém bater na mesa, o botão gira um pouco e vira 51%. Se esquentar, a resistência muda e vira 49%. É impreciso.
- **Computador Digital (Interruptor):** É um interruptor comum. Ou está ACESO ou APAGADO. Não importa se você aperta o botão com força ou devagar, o estado é claro e inequívoco. É fácil distinguir os dois estados.

3 3. Cópia e Degradação

Se você tirar uma xerox de uma xerox, e depois uma xerox da cópia... a imagem desaparece. Isso é o que acontece com dados analógicos (como fitas K7 antigas). O sinal degrada a cada manipulação.

No sistema digital, você pode copiar um arquivo 1 trilhão de vezes e a última cópia será **bit a bit idêntica** à original, porque estamos copiando a *ideia* do número (0 ou 1) e não a voltagem física exata.

4 4. Computadores Analógicos Existem?

Sim! Eles foram muito usados para controlar mísseis e simular física antes dos anos 70. Eles são incrivelmente rápidos para cálculo diferencial, mas:

1. São difíceis de programar (precisa recabear o circuito).
2. A precisão é limitada a cerca de 3 ou 4 casas decimais (devido ao ruído dos componentes).

Resumo: O computador usa circuitos elétricos (que são fisicamente analógicos), mas aplica uma "máscara" digital sobre eles para garantir que \$2 + 2\$ seja sempre \$4\$, e não \$3.99998\$ devido a uma oscilação na rede elétrica.

qual é conceito matemático que converter um sinal analógico para digital ?

O conceito matemático fundamental por trás da conversão de um sinal analógico (contínuo) para digital (discreto) chama-se **Discretização**.

Como o mundo real é contínuo (existem infinitos valores entre 1 segundo e 2 segundos, ou entre 1 Volt e 2 Volts), a matemática precisa "quebrar" essa continuidade em pequenos pedaços finitos que o computador consiga armazenar.

Esse processo de discretização acontece em duas etapas matemáticas distintas: uma no tempo (Eixo X) e outra na amplitude/valor (Eixo Y).

5 1. Amostragem (Sampling) - Discretização do Tempo (Eixo X)

Imagine que você tem uma curva desenhada num papel. A amostragem é o ato de olhar para essa curva apenas em intervalos de tempo específicos e ignorar o que acontece entre eles.

- **Matemática:** Em vez de ter uma função $f(t)$ contínua para todo t , você passa a ter uma sequência de números: $f(t_0), f(t_1), f(t_2)...$
- **A Regra de Ouro (Teorema de Nyquist-Shannon):** Este é o conceito matemático mais importante aqui. Ele diz que, para converter um sinal sem perder a informação original, você deve "fotografar" (amostrar) o sinal com uma frequência **pelo menos duas vezes maior** que a frequência máxima contida nele.
 - *Exemplo:* O ouvido humano ouve até 20.000 Hz (20 kHz). Para gravar música digitalmente (como num CD), a taxa de amostragem matemática deve ser no mínimo 40.000 Hz. Por segurança, o padrão do CD é 44.100 Hz.

6 2. Quantização (Quantization) - Discretização da Amplitude (Eixo Y)

Depois de "fotografar" o sinal em instantes específicos, você tem um valor de voltagem preciso (ex: 3.141592... Volts). Mas o computador não tem precisão infinita; ele tem um número limitado de bits (ex: 8 bits, 16 bits).

A quantização é o processo matemático de **arredondamento**. Você força o valor real a se encaixar no "degrau" mais próximo disponível na sua escala digital.

- **Matemática:** É um mapeamento de um conjunto infinito de valores (reais) para um conjunto finito de valores (inteiros).
- **Erro de Quantização:** A diferença entre o valor real (3.1415...) e o valor salvo (3.14) cria um pequeno erro matemático que, no mundo real, é ouvido como um chiado de fundo ("ruído de quantização"). Quanto mais bits você usa, menor é esse erro.

7 Resumo do Processo (PCM)

A junção desses dois conceitos forma o **PCM (Pulse Code Modulation)**, que é o padrão de áudio digital não comprimido (WAV, CD):

1. **Sinal Original:** Onda suave e perfeita.

2. **Amostragem:** Transforma a onda em "palitos" verticais a cada milissegundo.
3. **Quantização:** Arredonda a altura desses palitos para o número inteiro mais próximo.
4. **Codificação:** Transforma esses números inteiros em binário (010101).

aplicada as transformada de farrir como identifica o topo de frequência do sinal digital ?

A **Transformada de Fourier** (especificamente a **FFT** - Fast Fourier Transform, quando falamos de computadores) é a ferramenta matemática que permite olhar para um sinal digital e dizer: *"Quais frequências compõem este sinal?"*.

Para identificar o "topo" (o pico) de frequência, ou seja, a **frequência dominante** (aquele com maior energia ou volume), você não procura no sinal original (que é um desenho de onda no tempo). Você aplica a transformada e procura o valor mais alto no resultado.

Aqui está o passo a passo conceitual e matemático:

8 1. A Mudança de Perspectiva (Domínio do Tempo \$\\to\$ Frequência)

- **Antes da Transformada (Sinal Digital):** Você tem uma lista de valores de amplitude ao longo do tempo. Ex: [0.1, 0.5, 0.9, 0.4, -0.2...]. Olhando para isso, é difícil saber a frequência.
- **Depois da Transformada (Espectro):** A FFT pega essa lista e cria uma **nova lista**.
 - O **índice** dessa nova lista representa a **Frequência**.
 - O **valor** nessa lista representa a **Intensidade (Magnitude)** daquela frequência.

9 2. O Processo Matemático para achar o Topo

Para achar a frequência dominante (o topo), o computador segue este algoritmo:

Passo A: Calcular a Transformada (FFT)

O computador aplica o algoritmo FFT nas amostras digitais. O resultado são números complexos ($a + bi$).

Passo B: Calcular a Magnitude (O "Volume")

Como o resultado é complexo, precisamos saber o "tamanho" desse número para saber a força da frequência. Usamos o Teorema de Pitágoras (Módulo):

$$\text{Magnitude} = \sqrt{a^2 + b^2}$$

Agora temos um gráfico onde o eixo X é a frequência e o eixo Y é a magnitude.

Passo C: Encontrar o Valor Máximo (Peak Detection)

O computador varre essa nova lista de Magnitudes e encontra o **maior valor**.

- Se o maior valor estiver no índice k da lista, então k é o "bin" (cesto) onde está a frequência dominante.

10 3. A Fórmula de Conversão (O Pulo do Gato)

Achar o índice k não basta. Precisamos saber quantos **Hertz (Hz)** esse índice representa. Para isso, usamos a fórmula que conecta a matemática com o mundo físico (Taxa de Amostragem):

$$\text{Frequência} = \frac{\text{Índice do Pico (k)}}{\text{Número Total de Amostras}} \times \text{Taxa de Amostragem}$$

Exemplo Prático:

- Você gravou um áudio com Taxa de Amostragem de **44.100 Hz**.
- Você pegou um pedaço de **1.024 amostras** para analisar.
- A FFT gerou uma lista e você viu que o valor mais alto (o pico) está na posição (índice) **10**.

$$\text{Frequência} = \frac{10}{1.024} \times 44.100 \approx 430 \text{ Hz}$$

O computador agora sabe que o som predominante é uma nota de 430 Hz.

11 Resumo Visual

Imagine um equalizador de som daquele antigos.

1. As barras pulando para cima e para baixo são a Transformada de Fourier acontecendo em tempo real.
2. Identificar o "topo da frequência" é simplesmente olhar **qual barra está mais alta** naquele momento. A posição dessa barra (esquerda = grave, direita = agudo) diz a frequência.

Claro, Alonso. Vou **reformular seu resumo inteiro**, deixando **organizado, técnico, claro e direto**, mas sem perder nenhum conteúdo importante.

Também mantengo a linguagem adequada para prova/trabalho de Arquitetura de Computadores.

12 RESUMO REFORMULADO E ORGANIZADO

13 1. Execução da instrução `add $t0, $t1, $t2` (Tipo-R)

A instrução `add` é do tipo **R**, ou seja, opera somente com registradores e **não acessa a memória de dados**.

14 Fluxo de Execução

1. **Busca (IF)**
 - o O PC envia o endereço da instrução para a memória.
 - o A instrução (32 bits) é carregada no barramento.
 - o PC é incrementado ($PC + 4$).
2. **Decodificação (ID)**
 - o A Unidade de Controle lê o opcode `000000` → reconhece uma instrução **R**.
 - o Sinais ativados:
 - **RegDst = 1** → registrador destino está em rd (bits 15–11 → $$t0$).
 - **RegWrite = 1** → haverá escrita no banco de registradores.
 - **ALUSrc = 0** → segundo operando vem do registrador rt ($$t2$).
 - o Banco de registradores lê:
 - A = valor de $$t1$
 - B = valor de $$t2$
3. **Execução (EX)**
 - o A ULA recebe A = $$t1$ e B = $$t2$.
 - o Campo *funct* define a operação “`add`”.
 - o Resultado da soma sai na saída da ULA.
4. **Memória (MEM)**
 - o **MemRead = 0** e **MemWrite = 0** → memória não é usada.
5. **Write Back (WB)**
 - o Multiplexador MemtoReg = 0 → seleciona o valor da ULA.
 - o Escreve o resultado em $$t0$.

15 Resumo (ADD vs. SW)

- **ADD:** soma valores de registradores e escreve em registrador.
- **SW:** usa imediato, calcula endereço e escreve na memória, não altera registradores.

16 2. Execução da instrução `sw $t1, 104($t0)`

17 Fluxo Resumido

1. **Fetch** da instrução.
2. **Decode:**

- Lê \$t0 (base) e \$t1 (dado a ser gravado).
 - Unidade de Controle ativa:
 - **MemWrite = 1**
 - **ALUSrc = 1**
 - **RegWrite = 0**
3. **Execução (EX): cálculo do endereço**
- ULA recebe:
 - A = \$t0
 - B = imediato estendido (104)
 - ULA soma: endereço = \$t0 + 104
4. **Memória (MEM)**
- Endereço = saída da ULA
 - Dado = \$t1
 - **MemWrite = 1 → grava na RAM**
5. **WB**
- Não ocorre (RegWrite = 0).
-

18 3. Execução da instrução $lw \$t1, 104(\$t0)$

19 Fluxo

1. Igual ao SW até o cálculo do endereço.
2. **MemRead = 1 →** lê a RAM no endereço \$t0 + 104.
3. **WB:**
 - MemtoReg = 1 → dado vem da memória.
 - RegWrite = 1 → escreve em rt (\$t1).
 - RegDst = 0 → destino é rt.

O LW usa todos os estágios do uniciclo, sendo a instrução mais longa.

20 4. Flags do BEQ em um MIPS Uniciclo

A instrução **BEQ** compara dois registradores e, se forem iguais, altera o PC.

21 Sinais usados

- **ALUSrc = 0 →** ULA recebe dois registradores.
- **ALUOp = 01 →** operação = subtração.
- **Branch = 1 →** habilita lógica de desvio.
- **Zero** (saída da ULA) → fica 1 quando rs == rt.
- **PCSrc** → seleciona PC + 4 ou o endereço desviado.

A lógica funciona assim:

- A ULA calcula $rs - rt$.
- Se resultado = 0 → **Zero = 1**.

-
- O sinal **Branch** AND Zero → habilita a mudança de PC.
 - O endereço alvo é $(PC+4) + (imediato \ll 2)$.
-

22 5. Suporte a NOR, NAND e NOT com apenas 2 novos componentes

Para adicionar NOR, NAND e NOT:

23 Novos componentes

1. Inversor na entrada A.
2. MUX controlado por **AInvert** antes da entrada A.

O lado B já possui BInvert + Mux.

24 Seletores para cada operação

Instrução	AInvert	BInvert	Operação
NOR	1	1	AND
NAND	1	1	OR
NOT	1	1 (com B=0)	AND

25 6. RISC vs CISC

26 RISC

- Instruções simples e de tamanho fixo.
- Execução rápida e pipeline eficiente.
- Muitas instruções simples → menos microcódigo.
- Exemplos: **MIPS, ARM, SPARC, RISC-V**

27 CISC

- Instruções complexas (até múltiplos ciclos).
 - Menos instruções, mas mais poderosas.
 - Presença de microprogramação.
 - Exemplos: **x86, VAX, 68000**
-

28 7. Máquina de Von Neumann

- Memória única para **dados e instruções**.
- CPU composta por:
 - ULA

- Unidade de Controle
 - Registradores
 - Ciclos: **Fetch → Decode → Execute → Store**
 - Problema: **Gargalo de von Neumann** (um barramento único para tudo).
-

29 8. MIPS Multiciclo – FSM e contagem de ciclos

30 Quantidade de ciclos

- **LW:** 5 ciclos
- **ADD:** 4 ciclos
- **BEQ:** 3 ciclos
- **SW:** 4 ciclos

Total = 16 ciclos

31 9. Diferença entre Uniciclo e Multiciclo

32 Uniciclo

- 1 instrução = 1 ciclo.
- Clock muito longo.
- Hardware duplicado (não pode reutilizar).

33 Multiciclo

- Divide a instrução em vários ciclos menores.
 - Reutiliza hardware.
 - Clock curto.
 - Algumas instruções levam 3, outras 4 ou 5 ciclos.
-

34 ? Pergunta Final do Usuário:

“Sendo um RISC de 8 bits, como ele armazena o resultado de multiplicação?”

Se quiser, posso responder agora — basta confirmar!