Arquitetura RISC-V

Geral

- Memória endereçada a byte
- Instruções não operam diretamente na memória
  - Arquitetura Load/Store
    - Carrega nos registradores, altera, salva na memória
- Endianness [1] -> little-endian

Registradores

Registrador	Nome ABI	Descrição	Preservado em toda a chamada?
x0	zero	Hard-wired zero	—
x1	ra	Endereço de retorno	Não
x2	sp	Ponteiro de pilha	Sim
x3	gp	Ponteiro global	—
x4	tp	Ponteiro de Thread	—
x5	t0	Registrador de link temporário/alternativo	Não
x6–7	t1–2	Temporários	Não
x8	s0/fp	Registrador salvo/Ponteiro de quadro	Sim
x9	s1	Registrador salvo	Sim
x10–11	a0–1	Argumentos de função / valores de retorno	Não
x12–17	a2–7	Argumentos de função	Não
x18–27	s2–11	Registradores salvos	Sim
x28–31	t3–6	Temporários	Não
f0–7	ft0–7	Temporários FP	Não
f8–9	fs0–1	Registradores salvos FP	Sim
f10–11	fa0–1	Argumentos e valores de retorno FP	Não
f12–17	fa2–7	Argumentos FP	Não
f18–27	fs2–11	Registradores salvos FP	Sim
f28–31	ft8–11	Temporários FP	Não

Estrutura

- Segmentos
  - Segmento de dados (.data)
    - .align <>
    - .asciz str
    - .word w1,...,wn
    - .double, .float, .byte, .half
  - Segmento de texto (.text)
    - .align 2
    - .globl main
- Memória
  - Pilha cresce para baixo[2], heap para cima
- Alinhamento
  - .align x -> alinha para 2^x
  - Apesar de ser endereçado a byte [3], ele lê por palavra (4 bytes na arquitetura de 32 bits), então queremos alinhar nossos dados corretamente
  - Alinhamos o código com .align 2, pois cada instrução tem 32 bits (PC = PC + 4)

E/S

- Através de ecalls (chamadas ao sistema) (passar código por a7)

Name	Number	Description	Inputs	Ouputs
PrintInt	1	Prints an integer	a0 = integer to print	N/A
PrintFloat	2	Prints an floating point number	fa0 = float to print	N/A
PrintDouble	3	Prints a double precision floating point number	fa0 = double to print	N/A
PrintString	4	Prints a null-terminated string to the console	a0 = the address of the string	N/A
ReadInt	5	Read an int from input console	N/A	a0 = the int
ReadFloat	6	Read a float from input console	N/A	fa0 = the float
ReadDouble	7	Reads a double from input console	N/A	fa0 = the double
ReadString	8	Reads a string from the console	a0 = address of input buffer a1 = maximum number of characters to read	N/A
Sbrk	9	Allocate heap memory	a0 = amount of memory in bytes	a0 = address to the allocated block
Exit	10	Exits the program with code 0	N/A	N/A
PrintChar	11	Prints an ascii character	a0 = character to print (only lowest byte is considered)	N/A
ReadChar	12	Read a character from input console	N/A	a0 = the character
Exit2	93	Exits the program with a code	a0 = the number to exit with	N/A

Instruções

- Load (Registrador <- Memória)
  - l
    - w, b, h
      - lw r_d, des(r_s)
    - a (pseudoinstrução do RARS para address)
      - la r_d, label
- Store (Memória <- Registrador)
  - sw r_s, des(r_d)
- Register-register
  - add, sub
  - slt
  - and, or, xor
  - sll, srl, sra
  - (mul, div)
- Register-immediate
  - addi
  - slti
  - andi, ori, xori
  - slli, srli, srai
- Jump
  - jal
  - jr
  - j label (pseudo)
- Branch
  - beq, bne
  - blt
  - bge

Procedimentos

- Ideia

    - Parâmetros de entrada em local padrão para o procedimento
    - Transfere controle
    - Realiza tarefa
    - Valor de retorno em local padrão para o programa
    - Retoma controle
- Convenção
    - a0-a7: argumentos de funções e valores de retorno
    - ra: endereço de retorno

Pilha

- A cada chamada
    - addi sp, sp, -bytes
    - sw reg, des(sp)
- Quando volta
    - lw reg, des(sp)
    - addi sp, sp, bytes

CPU Monociclo

ISA

Informações Gerais

- 32 registradores
- 6 formatos de intruções (tipos)

Classes das instruções

- Operações aritméticas (ex. add)
- Operações lógicas (ex. and)
- Interação com memória (ex. lw)
- Controle de fluxo (ex. beq)
- Comparação (ex. slt)

Tipos

- R -> Registradores
- I -> Imediato
- S -> Store
- B -> Branch
- U -> Upper
- J -> Desvio

Formatos

Name (Field size)	Field						Comments
	7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

Deslocamento

- Algumas instruções usam deslocamento com base (ex. 0(sp)), outras, como branch, usam deslocamento baseado no PC. Esses offsets ficam no imm
    - Para jumps, arquitetura coloca só um 0 no final do imm (shift pra esquerda) ao invés de dois, pois podemos ter instruções de 16 bits

Arquitetura RISC-V

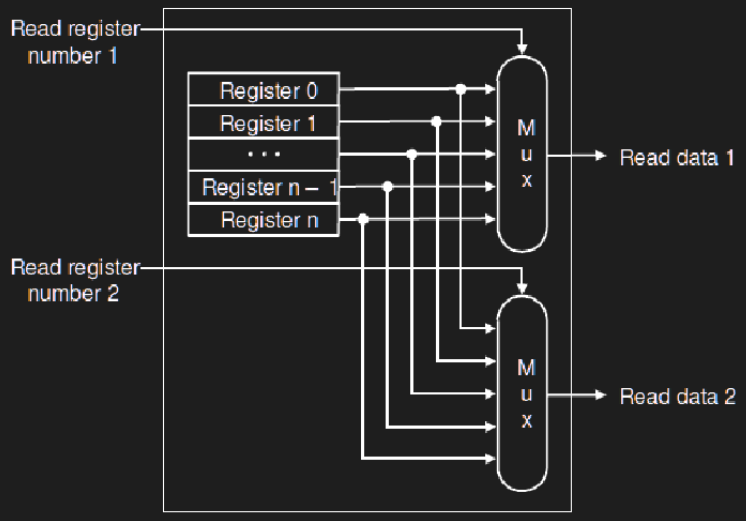
Implementações

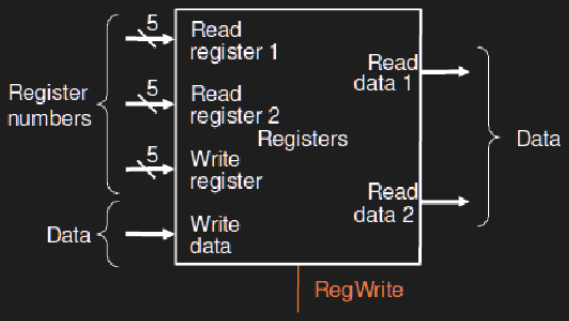
- Monociclo -> instrução em 1 ciclo de clock
- Multiciclo -> instrução em x ciclos de clock
- Pipelined -> instrução em x ciclos de clock + partes de várias instruções executadas simultaneamente

Componentes

Banco de registradores

- Mux para selecionar quais registradores serão lidos/escritos
- Flag RegWrite para indicar escrita

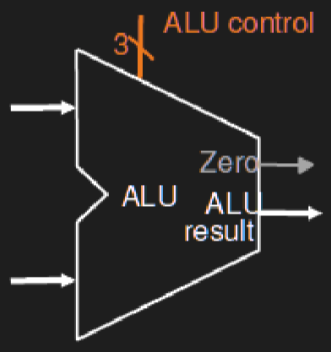




Unidade de Controle

ULA

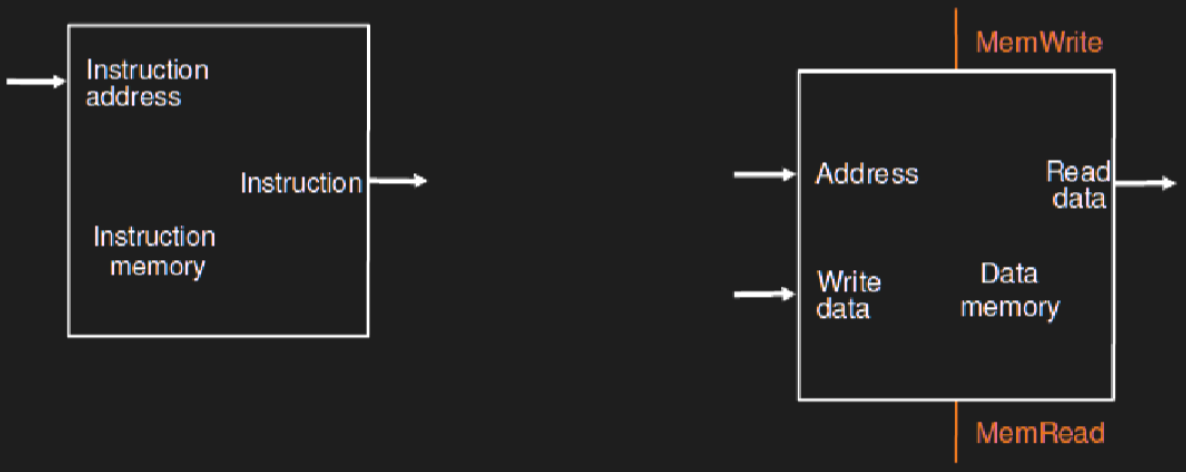
- Recebe ALUControl (formado por ALUOp (vindo da UC), func3 e func7)



Memória

- Dividida em duas (para evitar conflitos estruturais): instrução e dados
- Para dados
  - Flags MemWrite e MemRead
  - Recebe o endereço da ULA
- Para instrução
  - Recebe o endereço do PC

Instrução Dados

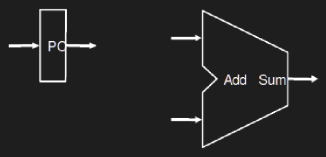


-

Entrada/Saída

+ PC e somador

PC Somador

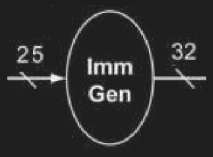


-

+ ImmGen (geração do valor imediato e extensão de sinal)

- Recebe também ImmSrc da UC

Geração do valor imediato
e extensão de sinal



-

+ vários multiplexadores

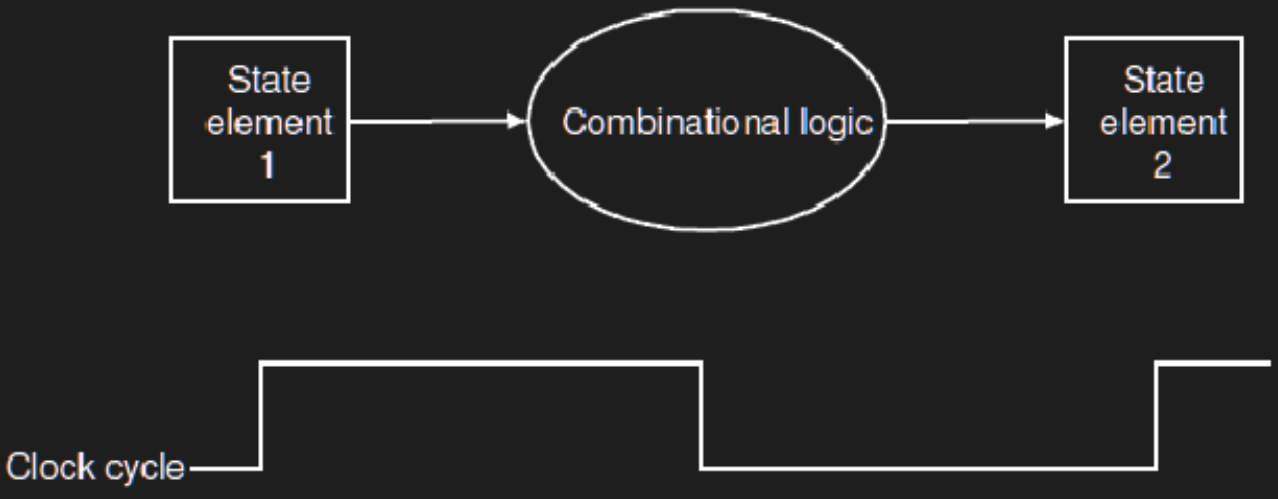
- Escolher se PC vai ser valor imediato do jump ou PC+4
- Escolher se a ULA vai receber valor imediato ao de registrador
- Escolher se vai escrever em registrador um resultado da ULA ou um valor lido da memória

Interconexão

- Por linhas de dados (caminho de dados) e linhas de controle (UC - a depender da fase e da instrução)

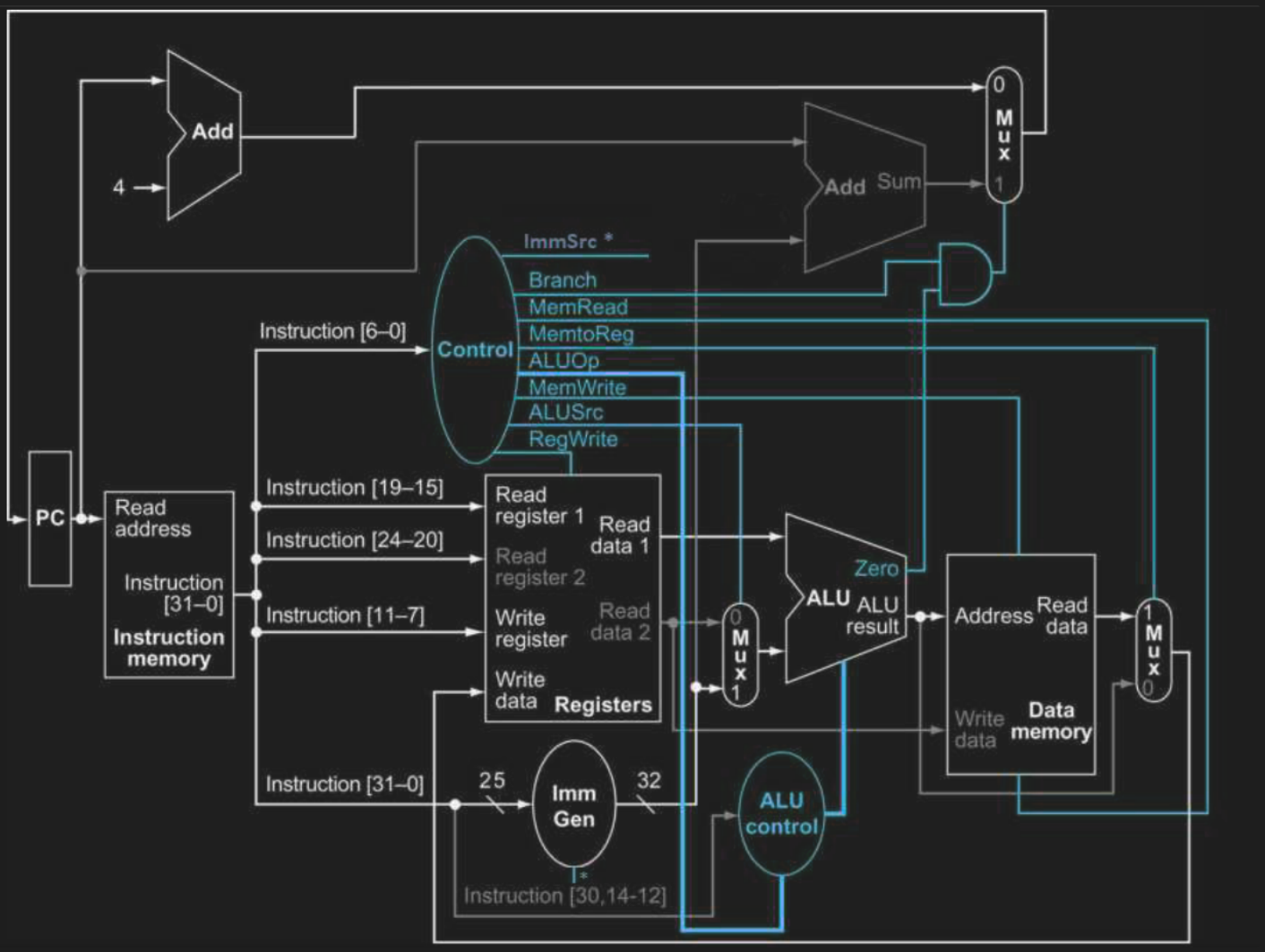
Metodologia de Clocking

- Valores de elementos lógicos sequenciais são atualizados na transição de clock
  - Evita feedback e condições de corrida



-

Implementando Tipos de Instrução



-

Tipo R

- Faz
  - Lê dois registradores
  - Realiza operação aritmética ou lógica
  - Escreve em um registrador
- Usa
  - Banco de registradores
  - ULA
- Flags
  - RegWrite
  - ALUOp -> operação

- ALUSrc -> registrador

Tipo I

- Faz
  - Lê um registrador
  - Tem um valor imediato
  - Realiza operação aritmética ou lógica
  - Pode ler memória (ex. lw)
  - Escreve em um registrador
- Usa
  - Banco de registradores
  - ImmGen
  - ULA
  - Memória
- Flags
  - RegWrite
  - ALUOp -> operação ou adição
  - ALUSrc -> imediato
  - MemRead
  - MemToReg

Tipo S

- Faz
  - Lê dois registradores (base e origem)
  - Tem um valor imediato (deslocamento)
  - Realiza operação aritmética ou lógica (adição)
  - Escreve na memória
- Usa
  - Banco de registradores
  - ImmGen
  - ULA
  - Memória
- Flags
  - ALUOp -> adição
  - ALUSrc -> imediato
  - MemWrite

Tipo B

- Faz
  - Lê dois registradores
  - Tem um valor imediato (offset)
  - Realiza operação aritmética
- Usa
  - Banco de registradores
  - ImmGen
  - ULA
  - Somador
- Flags
  - ALUOp -> subtração
  - ALUSrc -> registrador
  - Branch
  - (Zero na ULA pro beq)

UC (Flags)

Instruction	Execution/address calculation stage control lines		Memory access stage control lines			Write-back stage control lines	
	ALUOp	ALUSrc	Branch	Mem-Read	Mem-Write	Reg-Write	Memto-Reg
R-format	10	0	0	0	0	1	0
lw	00	1	0	1	0	1	1
sw	00	1	0	0	1	0	X
beq	01	0	1	0	0	0	X

-
-
  - ImmSrc

ALUControl

- Operação determinada por ALUOp (UC) + func3 + func7
- ALUOp escolhe se operação
  - 00 (Adição) -> Instruções de load e store
  - 01 (Subtração) -> beq
  - 10 (Depende) -> pega de func3 e func7

Unidade de Controle da ULA

ALU control lines	Function
000	AND
001	OR
010	add
110	subtract

Instruction opcode	ALUOp	Operation	Funct7 field	Funct3 field	Desired ALU action	ALU control input
lw	00	load word	XXXXXXX	XXX	add	010
sw	00	store word	XXXXXXX	XXX	add	010
beq	01	branch if equal	XXXXXXX	XXX	subtract	110
R-type	10	add	0000000	000	add	010
R-type	10	sub	0100000	000	subtract	110
R-type	10	and	0000000	111	AND	000
R-type	10	or	0000000	110	OR	001

-

ImmGen

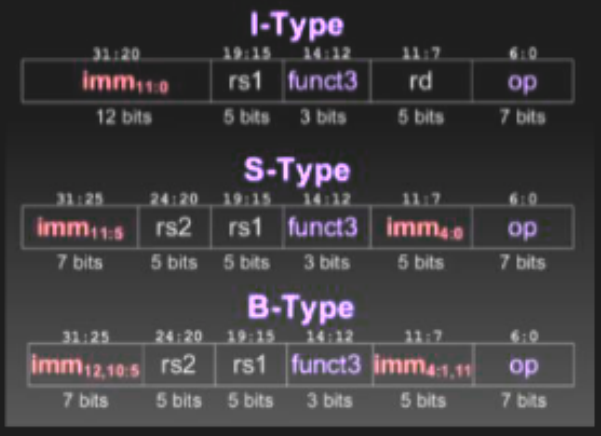
- Da onde pegar valor imediato depende de ImmSrc (UC)

Elaboration: The immediate generation logic must choose between sign-extending a 12-bit field in instruction bits 31:20 for load instructions, bits 31:25 and 11:7 for store instructions, or bits 31, 7, 30:25, and 11:8 for the conditional branch. Since the input is all 32 bits of the instruction, it can use the opcode bits of the instruction to select the proper field. RISC-V opcode bit 6 happens to be 0 for data transfer instructions and 1 for conditional branches, and RISC-V opcode bit 5 happens to be 0 for load instructions and 1 for store instructions. Thus, bits 5 and 6 can control a 3:1 multiplexor inside the immediate generation logic that selects the appropriate 12-bit field for load, store, and conditional branch instructions.

-

Unidade de geração do valor imediato

$ImmSrc_{1:0}$	ImmExt	Instruction Type
00	{{20{instr[31]}}, **instr[31:20]**}	I-Type
01	{{20{instr[31]}}, **instr[31:25], instr[11:7]**}	S-Type
10	{{19{instr[31]}}, **instr[31], instr[7], instr[30:25], instr[11:8], 1'b0**}	B-Type



-

Pipeline

Arquitetura RISC

- Problemas da CISC
  - Instruções raramente usadas -> maior tempo para decodificar e executar
  - IPC (instruction per cycle) menor que 1
  - Mais difícil implementar pipeline
  - UC microprogramada

Soluções

- Instruções simples -> número fixo de ciclos para executar (usualmente 1)
- Poucos formatos de instrução -> facilita o pipeline
- Arquitetura orientada a registrador
- Acesso à memória principal só por load/store
- UC hardwired

Desafios da RISC

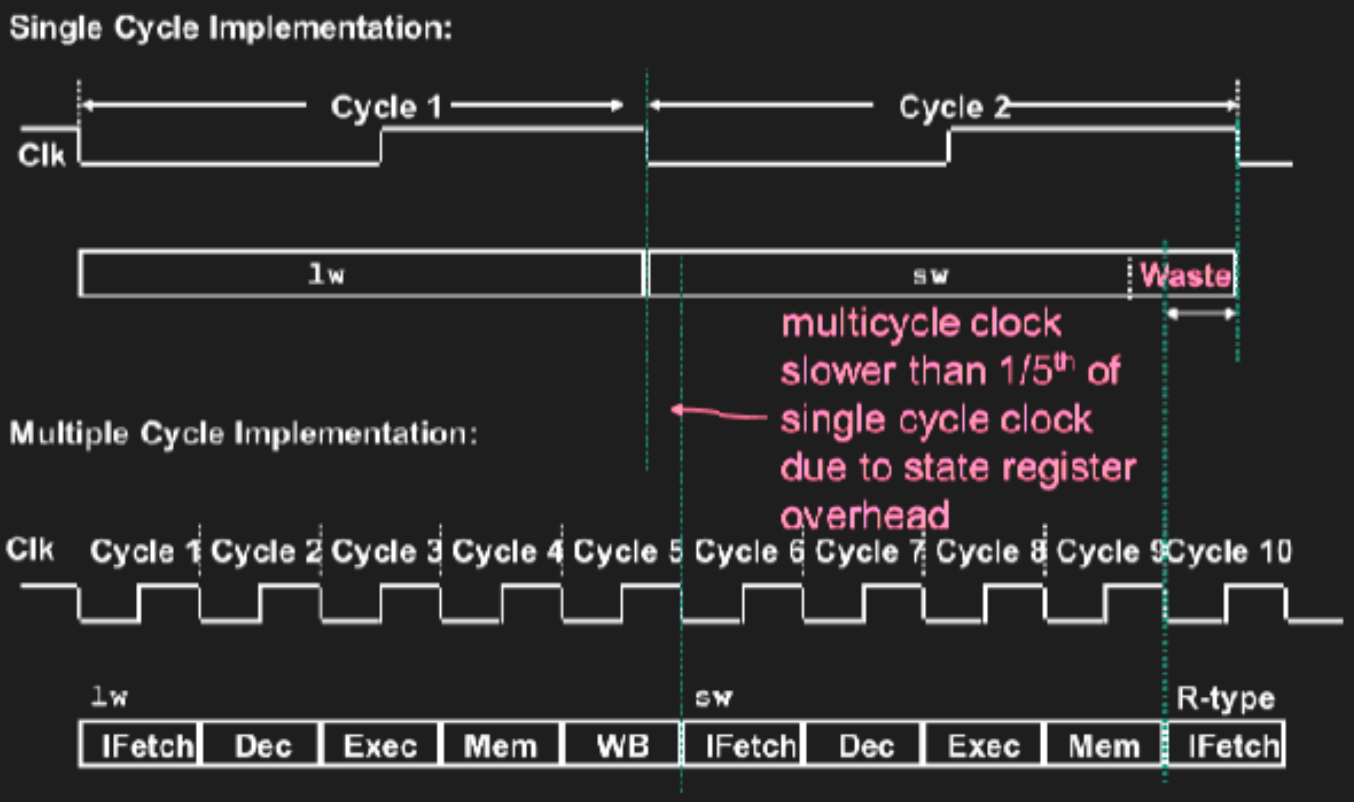
- Lento -> ciclo de clock deve ser do tamanho da instrução mais lenta

Multiciclo

- Instrução demora mais de um ciclo de clock para executar
  - Cada passo é um ciclo de clock

Vantagens

- Rápido -> nem todas as instruções tem o mesmo número de passos



Pipeline

- Computador com um único ciclo: 45 ns/ciclo * 1 CPI * 100 instruções = 4500 ns
- Computador com multiciclo: 10 ns/ciclo * 4,2 CPI * 100 instruções = 4200 ns
- Computador com 5 estágios ideais de pipeline: 10 ns/ciclo * (1 CPI * 100 instruções + 4 ciclos para esvaziar) = 1040 ns

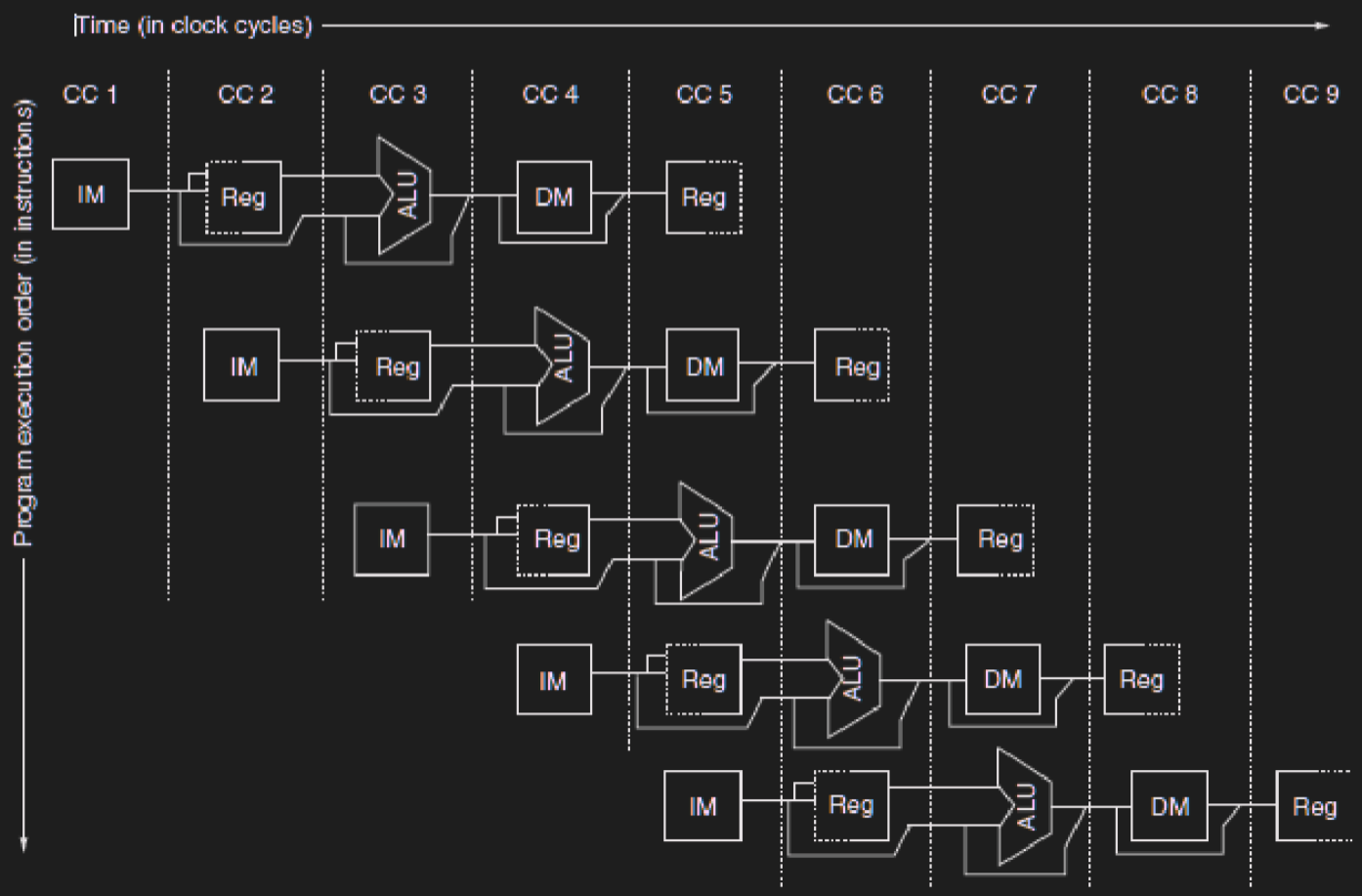
CPI: *cycle per instruction, clocks per instruction*

- Multiciclo + diferentes estágios executados em paralelo
- Considerações
  - Aumenta throughput
  - Limitado pelo estágio mais lento
  - Trava se houver dependências

Estágios

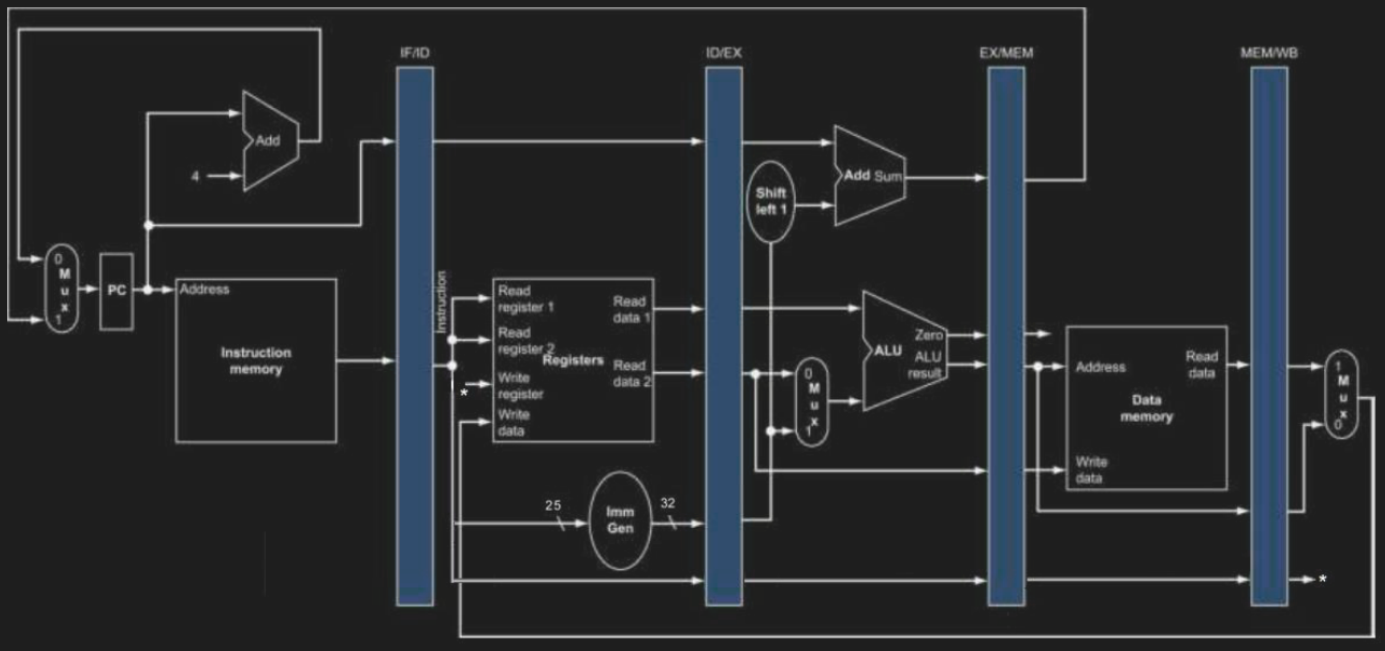
- IF (Instruction Fetch) -> Busca da instrução
  - Usa: Memória de instrução
- ID (Instruction Decode) -> Busca dos operandos + Decodificação da instrução
  - Usa: Banco de registradores
  - UC fica aqui
- EX (Execute) -> Executa operação na ULA ou calcula endereço de memória
  - Usa: ULA
- MEM (Memory) -> Acessa a memória principal
  - Usa: Memória de Dados
- WB (Write Back) -> Escreve nos registradores
  - Usa: Banco de registradores

Exemplo em função dos componentes usados

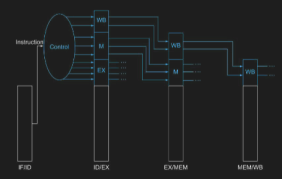


Registradores Intermediários

- Salvam saída de um estágio n para ser usada no próximo ciclo como entrada do estágio n+1



- Registradores
  - IF/ID
    - PC (base para futuro branch)
    - IR (instrução; para gerar sinais de controle, saber registradores usados e gerar valor imediato etc)
  - ID/EX



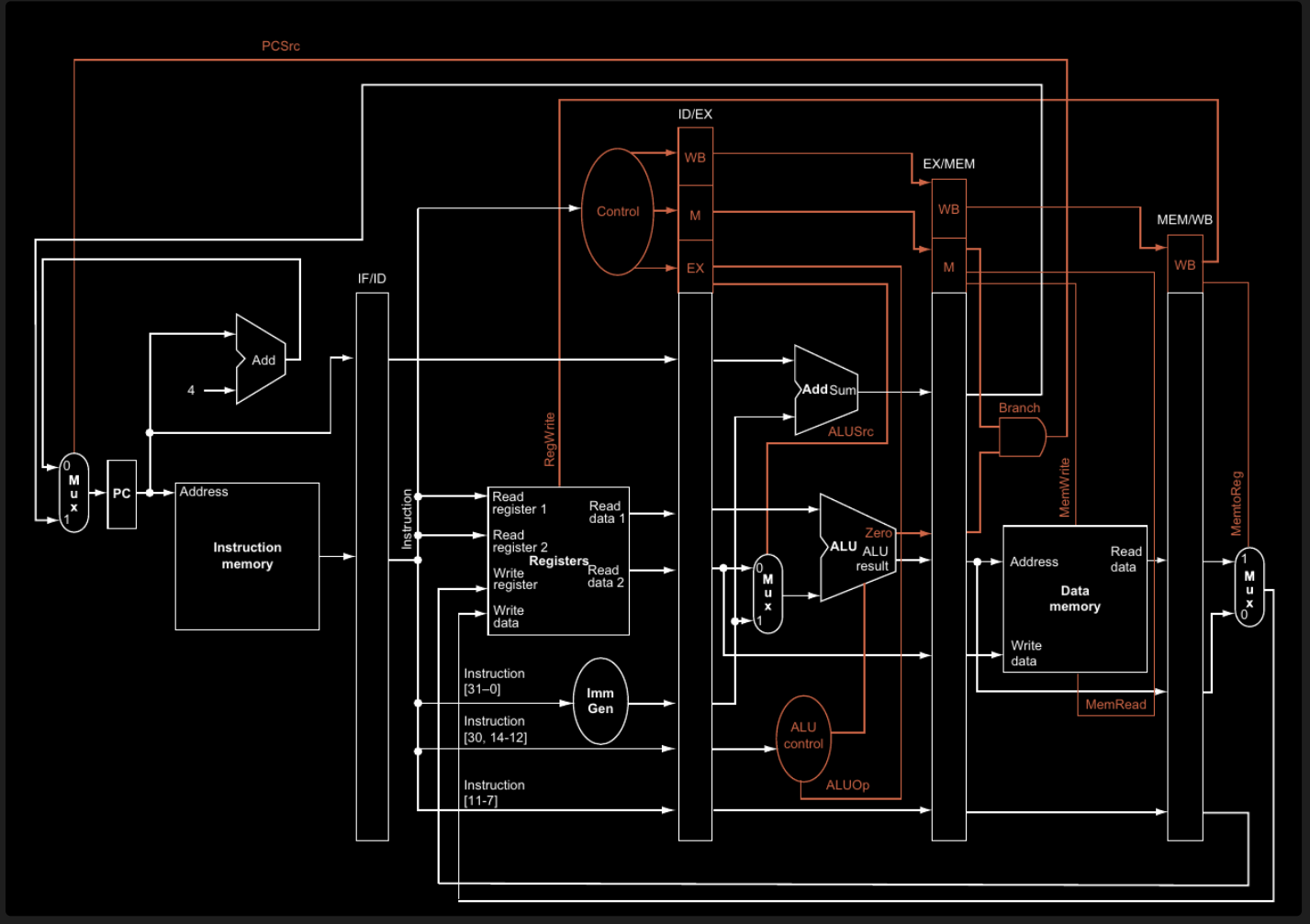
    - A (r1)
    - B (r2)

    - Imm
    - PC (base para futuro branch)
    - func3, func7 (parte delas pro ALUControl)
    - rd (endereço para futuro write-back)
- EX/MEM
    - ALUOutput
    - BranchTarget
    - B (dados para escrita futura na memória)
    - rd (endereço para futuro write-back)
    - zero (condição para futuro branch)
- MEM/WB
    - ALUOutput (dados para futuro write-back)
    - LMD (valor lido da memória; dados para futuro write-back)
    - rd (endereço para futuro write-back)

Sinais de Controle

ID/EX	EX/MEM	MEM/WB
ALUOp	MemRead	RegWrite
ALUScr	MemWrite	MemtoReg
MemRead	Branch	
MemWrite	RegWrite	
Branch	MemtoReg	
RegWrite		
MemtoReg		

Caminho de Dados com Sinais de Controle



Depêndencia de Dados

Alguns problemas

- Conflitos Estruturais
  - Mais de uma instrução acessando o mesmo componente
- Dependência de Dados
  - Instrução depende de valor ainda não escrito
- Dependência de Controle
  - Branch e Jump -> próxima instrução não é a logo em seguida

Conflitos Estruturais

Banco de Registradores

- Fases: ID e WB
- Solução: divisão de clock
  - Escrita (WB) na borda de descida, depois leitura na borda de subida

Memória

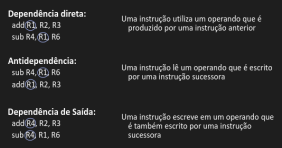
- Fases: IF e MEM
- Solução: separar em memória de dados e memória de instrução
  - (Mas o endereçamento é único)
  - E caches L1 separadas
- Hoje em dia as memórias já possuem vários acessos independentes

ULA

- Fases: IF e EX
- Solução: colocar um somador para a fase IF ao invés de usar a ULA

Dependência de Dados

Tipos



- Dependências verdadeiras
  - Dependência direta -> RAW (Read after write)
- Dependências falsas -> causadas por execução fora de ordem
  - Antidependência -> WAR (Write after read)
  - Dependência de saída -> WAW (Write after write)

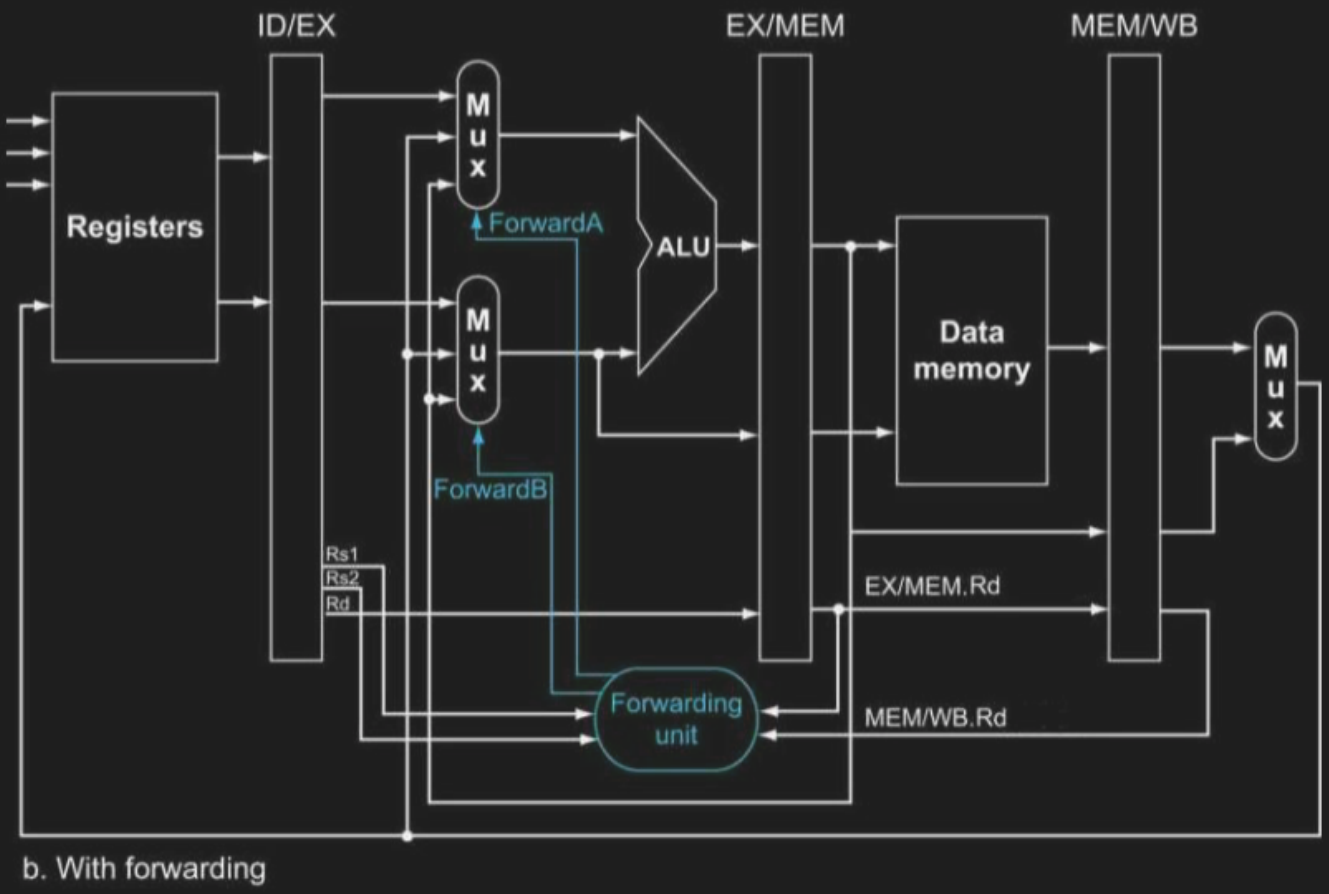
Hazard

- Dependências que fazem o pipeline parar -> bolha/interlock
  - A bolha não atrasa só aquela instrução, mas todas abaixo, pois não pode ter mais de uma instrução na mesma fase
    - ex. se atrasarmos o IF de uma instrução, não podemos começar outra, se não teríamos duas instruções no estágio IF ao mesmo tempo
  - Número de bolhas = número de ciclos que não finalizam instrução
- No nosso caso, só RAW é hazard

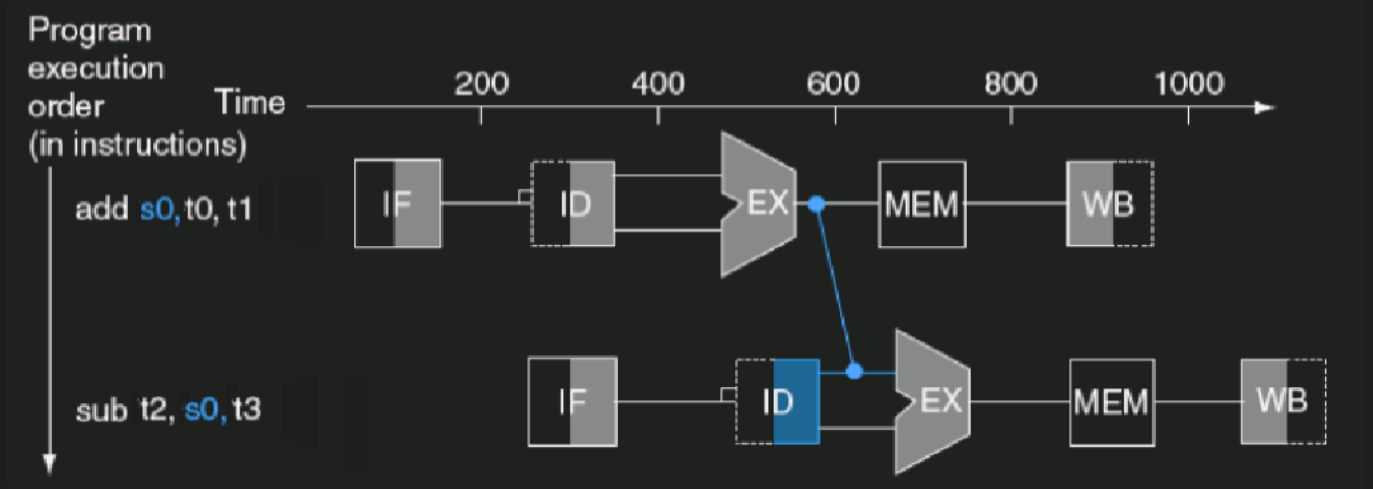
RAW

- Regra: é preciso duas instruções entre uma dependência RAW
  - Isso alinha os estágios de WB do write e ID do read. Por causa da divisão de clock, é garantido que o WB executa primeiro
  - Caso nenhuma: 2 bolhas
  - Caso uma: 1 bolha
- É possível manualmente reordenar instruções para diminuir o número de bolhas
  - Desde que não altere lógica, parece seguir a ordem: escrita, aritmética, leitura
- Se a arquitetura não tem parada, é necessário usar a instrução "nop"
- Como criar bolha? -> Não alterando registradores intermediários
  - IF -> Não atualiza PC
  - ID -> Não atualiza IF/ID
  - EX -> Desativa sinais de controle dos estágios EX, MEM e WB

Forwarding



-
- Resolve a maioria dos casos de RAW
- Ideia: ID ainda vai ler o valor desatualizado, mas mandamos o valor calculado atualizado para o EX ao invés do valor lido



-
- Implementação
  - Mux antes dos 2 valores mandados à ULA
    - 3 opções
      - Banco de registradores e/ou valor imediato (normal)
      - EX/MEM.ALUOutput (valor calculado na instrução anterior)
      - Mux[MEM/WB.ALUOutput, LMD] (valor calculado duas instruções anteriores)
        - A razão do mux é pra resolver uma dependência da lw que será comentada

Unidade de Forwarding

- Gera sinais de Forwarding (ForwardA e ForwardB)
  - 00 -> normal
  - 10 -> instrução anterior
  - 01 -> duas instruções anteriores
- Para a lógica interna, é necessário guardar rd, rs1 e rs2 no ID/EX
  - Recebe rd, rs1 e rs2 do ID/EX
  - Recebe rd do EX/MEM e MEM/WB
- Lógica interna

- **Dependência EX:**

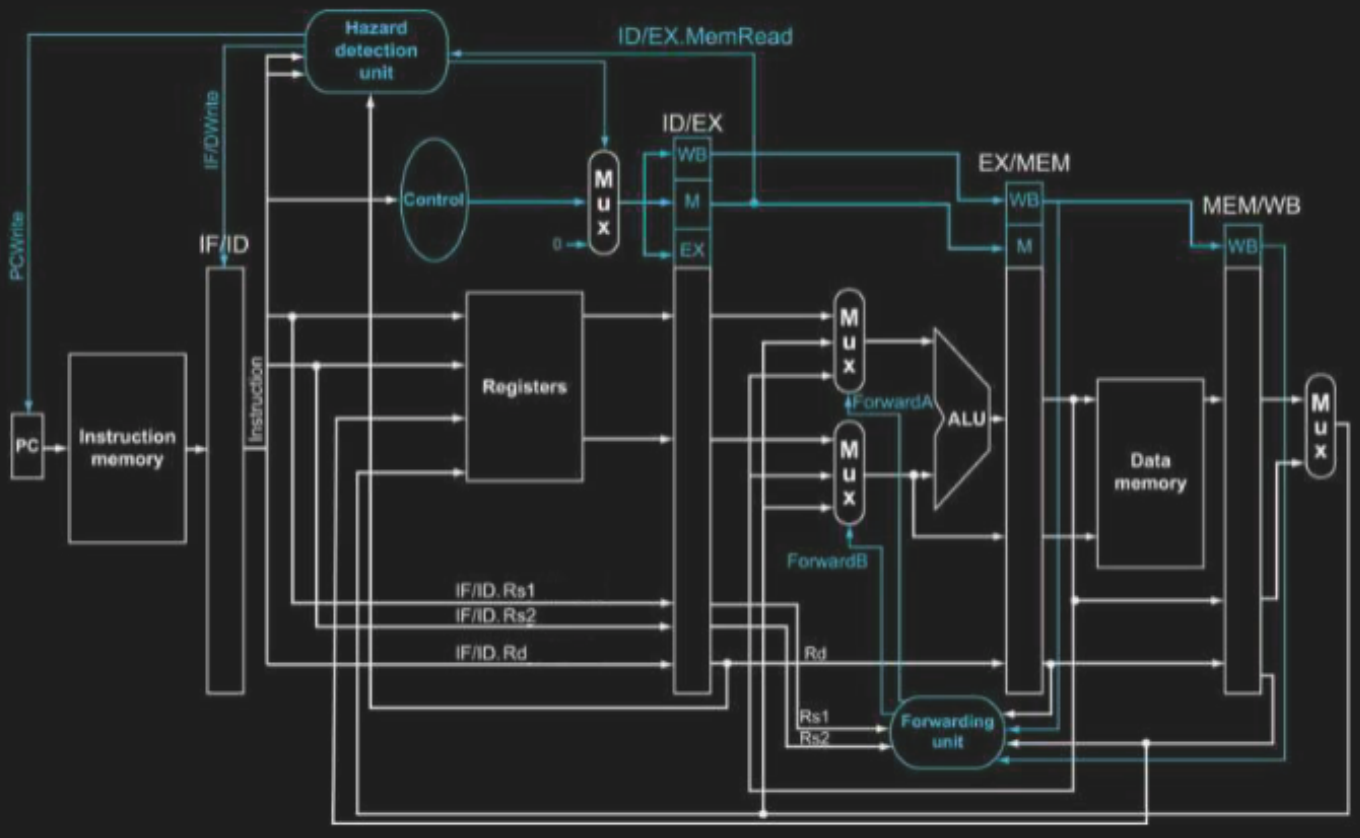
  if (EX/MEM.RegWrite
  and (EX/MEM.Rd ≠ 0)
  and (EX/MEM.Rd = ID/EX.Rs1)) ForwardA = 10

- **Dependência MEM:**

  if (MEM/WB.RegWrite
  and (MEM/WB.Rd ≠ 0)
  and (MEM/WB.Rd = ID/EX.Rs1)) ForwardA = 01

- Caso as duas sejam verdadeira, escolhe a EX (mais perto)

Unidade De Parada



- Forward não resolve todos os hazards
  - Exemplo: lw (valor lido da memória e escrito em registrador)
    - lw t8, 0(t0)
    - add t5, t8, t3
      - Se adiantar o valor calculado (ULA), t8=t0+0, o que é errado
- Unidade de parada insere o nop/bolha necessário
- Lógica interna

```
if (ID/EX.MemRead and                          // leitura da memória
        ((ID/EX.Rd = IF/ID.Rs1) or             // Destino (EX) == Origem 1 (ID)
                (ID/EX.Rd = IF/ID.Rs2)))       // Destino (EX) == Origem 2 (ID)
        stall the pipeline
```

References

---

1. Analogia "big end" e "little end" de um ovo. Big-endian começa a descascar pelo lado maior, little-endian pelo lado menor ↩
2. Quando dizemos que a stack "cresce para baixo", queremos dizer que cresce em direção a endereços menores. Por isso, temos que decrementar o sp e não incrementar. Link ↩

Figure 1: Byte addressing.

Figure 2: Word addressing.

3. Endereçado a byte = byte é a menor unidade que conseguimos ter acesso. Mas, por questões de performance, a palavra inteira é colocada na memória. Se tivermos um int (word), precisamos que esteja alinhado no começo de uma palavra, se não teremos um erro. Para strings, usamos .align 0, pois não temos essa limitação ↩