Capítulo 5 (Livro Texto P & H)

Tradução e Adaptação:
Ney Laert Vilar Calazans

(Versão sendo traduzida em 27/10/2009)

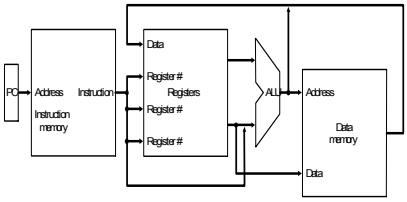
O Processador: Blocos de Dados e de Controle

- Estamos prontos para estudar uma implementação do MIPS
- Simplificado para conter apenas:
  - instruções de referência à memória: `lw, sw`
  - instruções lógicas e aritméticas: `add, sub, and, or, slt`
  - instruções de controle de fluxo: `beq, j` (não implementado!!)
- Implementação Genérica:
  - usa-se o contador de programa (program counter ou PC) para suprir o endereço da instrução
  - obtém-se a instrução da memória
  - lê-se os registradores fonte
  - usa-se o código da instrução para decidir exatamente o que fazer
- Todas as instruções usam a ULA após ler dos registradores fonte

  Por quê? Referências à memória? Aritmética? Fluxo de controle?

Mais Detalhes de Implementação

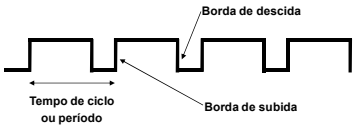
- Visão Abstrata/Simplificada:



- Dois tipos de unidades funcionais:
  - Elementos que operam sobre dados (combinacionais)
  - Elementos que armazenam informação de estado (sequenciais)

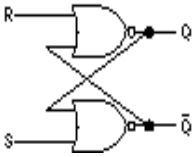
Elementos de Estado

- Sem relógio ou com relógio (Clock)
- Relógios são usados em lógica síncrona
  - quando se deve atualizar o conteúdo de um elemento que armazena estado?



Um elemento de estado sem relógio

- O *latch* set-reset
  - Saída depende das entradas atuais e também das entradas anteriores



Latches e Flip-flops

- A saída é igual ao valor armazenado dentro do elemento (não é necessário permissão para fazer acesso ao que existe no elemento)
- Mudanças de valor são condicionadas ao relógio:
  - Latches: sempre que entradas mudam, e relógio está ativo
  - Flip-flop: estado muda somente na borda de um relógio (método baseado em sensibilidade à borda)

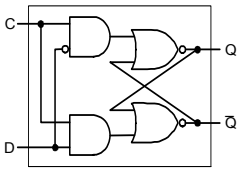
"logicamente ativo", — pode significar tensão alta ou tensão baixa

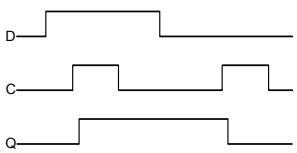
Um método baseado em sensibilidade 'a borda do relógio define quando sinais podem ser lidos ou escritos. Pressupostos:

1. Nenhum sinal de entrada varia durante a borda ativa do relógio!
2. Ninguém deve ler um sinal ao mesmo tempo em que ele está sendo escrito!

Latch D

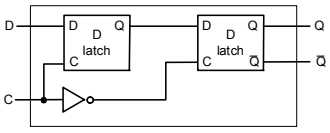
- Duas entradas:
  - O valor de dado a ser armazenado (D)
  - O sinal de relógio (C) indicando quando ler & armazenar D
- Duas saídas:
  - O valor do estado interno (Q) e seu complemento





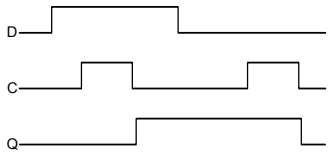
Flip-flop D

- Saída muda apenas durante uma das bordas do relógio



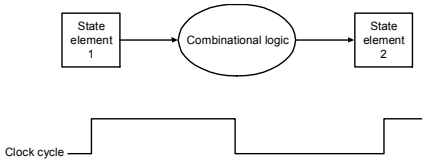
- *Flip-flop* Mestre-Escravo implementado com dois *latches* operando em fases opostas do mesmo relógio

- Funcionalidade do *Flip-flop* Mestre-Escravo



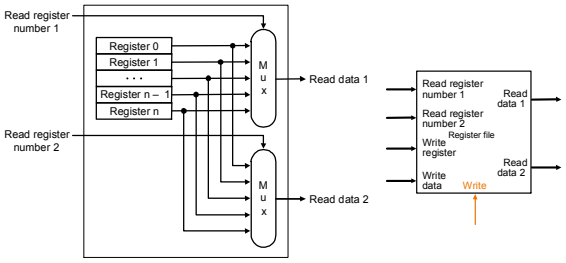
Nossa Implementação

- **Assume-se um método baseado em borda**
- **Execução típica:**
  - **Lê-se conteúdos de alguns elementos de estado,**
  - **Envia-se valores através de alguma lógica combinacional**
  - **Escreve-se os resultados em um ou mais elementos de estado**



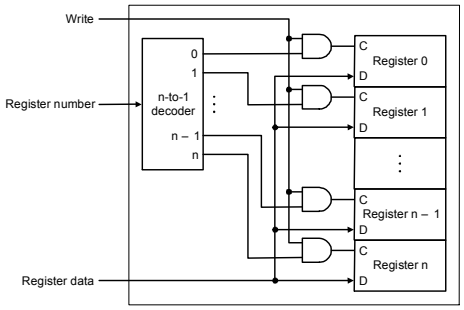
Banco de Registradores (Register File) - Leitura

- **Construído usando-se flip-flops D**



Banco de Registradores (Register File) - Escrita

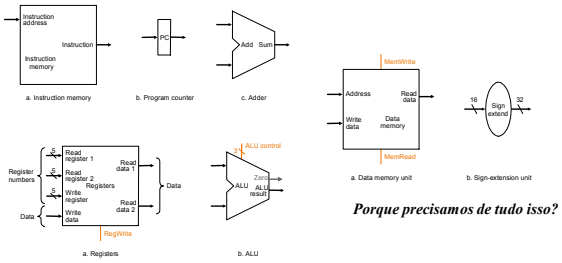
- **Notar: usa-se (de novo) o relógio para determinar quando escrever em um dos registradores (assume-se aqui sensibilidade à borda de subida)**



Ver Figuras 5.4-5.10 do livro-texto

Implementação Simples

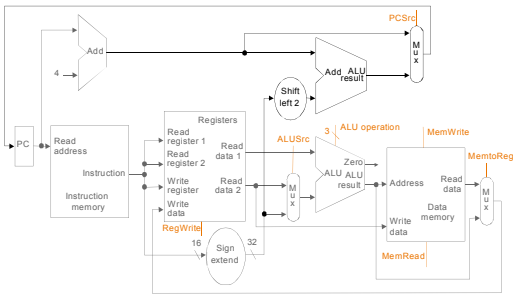
- **Inclui-se as unidades funcionais necessárias para cada instrução**



Porque precisamos de tudo isso?

Construindo o Bloco de Dados

- Usa-se multiplexadores para *costurar* os blocos funcionais juntos



Controle

- Seleciona as operações a executar (ULA, leituras/escritas, etc.)
- Controla o fluxo de dados (comandando os multiplexadores)
- A informação vem dos 32 bits da instrução
- Exemplo:

add $8, $17, $18 **Formato da Instrução (binário):**

000000	10001	10010	01000	00000	100000
op	rs	rt	rd	shamt	funct

- A operação da ULA é definida com base no tipo de instrução (opcode – bits 31-26) e no código de função (function – bits 5-0)

Bloco de Controle

- Por exemplo, que operação a ULA faz durante a execução da instrução lw $1, 100($2)? Formato da Instrução (decimal):

35	2	1	100
op	rs	rt	deslocamento de 16 bits

- Entrada de controle da ULA (exemplo)

```
000   AND
001   OR
010   add
110   subtract
111   set-on-less-than
```

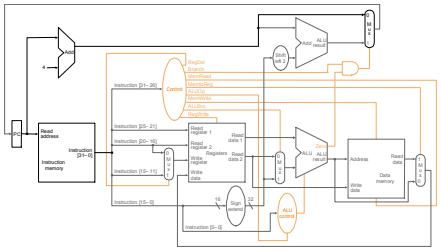
- Porque o código da subtração seria 110 e não 011?

Bloco de Controle

- Deve ser um hardware capaz de computar a entrada de controle de 3 bits da ALU
  - Dado o tipo de instrução
    00 = lw, sw
    01 = beq,
    11 = arithmetic
    → ALUOp é computado a partir do tipo de instrução (bits 31-26)
  - Código de função para operações aritméticas
- Descreve-se as operações usando uma tabela verdade (que vai ser implementada com portas lógicas):

ALUOp		Funct field						Cód Oper
ALUOp1	ALUOp0	F5	F4	F3	F2	F1	F0	
0	0	X	X	X	X	X	X	010
0	1	X	X	X	X	X	X	110
1	X	X	X	0	0	0	0	010
1	X	X	X	0	0	1	0	110
1	X	X	X	0	1	0	0	000
1	X	X	X	0	1	0	1	001
1	X	X	X	1	0	1	0	111

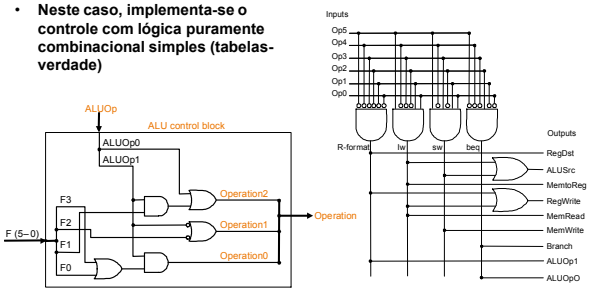
Bloco de Controle



Instruction	RegDst	ALUSrc	Memto-Reg	Reg Write	Mem Read	Mem Write	Branch	ALUOp1	ALUp0
R-format	1	0	0	1	0	0	0	1	0
lw	0	1	1	1	1	0	0	0	0
sw	X	1	X	0	0	1	0	0	0
beq	X	0	X	0	0	0	1	0	1

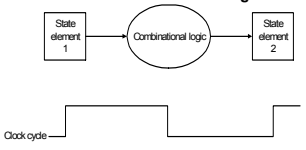
Bloco de Controle

- **Neste caso, implementa-se o controle com lógica puramente combinacional simples (tabelas-verdade)**



Nosso Bloco de Controle Simples

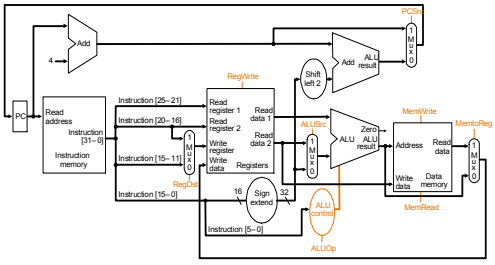
- **Toda a lógica é combinacional**
- **Espera-se que tudo estabilize, e a coisa certa é realizada**
  - **A ULA pode não produzir a "resposta correta" imediatamente**
  - **Usamos sinais de controle de escrita junto com o relógio para determinar quando escrever**
- **O período do relógio (*cycle time*) é determinado pelo tempo necessário ao caminho mais demorado da lógica combinacional**



Estamos ignorando alguns detalhes como os tempos de "setup" e "hold"

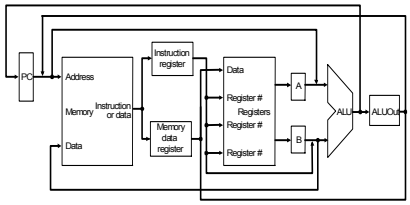
Implementação Monociclo Simples

- **Calcula-se o período de relógio assumindo que todos os atraos são desprezíveis, exceto:**
  - **O da memória (2ns), o da ULA e dos somadores (2ns), e o tempo de acesso a um registrador do banco de registradores (1ns)**



Onde Estamos Indo?

- **Problemas de implementações monociclo:**
  - **Que ocorre se temos instruções mais complexas, como as de ponto flutuante?**
  - **Desperdício de área**
- **Uma Solução:**
  - **usar um período "menor"**
  - **Fazer instruções diferentes gastarem diferentes números de ciclos**
  - **Um exemplo de Bloco de Dados "multiciclo":**

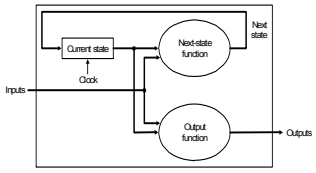


Abordagem Multiciclo

- **Reusamos as mesmas unidades funcionais:**
  - **ULA usada para computar endereços e para incrementar o PC**
  - **Memória usada para conter instruções e dados (organização von Neumann ao invés de Harvard)**
- **Os sinais de controle não são mais determinados unicamente pela instrução, pois será preciso escalonar sua geração ao longo dos ciclos.**
  - **Por exemplo, o que a ULA deve fazer para a instrução de "subtração", e em que ciclo da execução desta se "ativa" a ULA?**
- **Assim, deve-se usar uma máquina de estados finita para implementar o Bloco de Controle**

Revisão: Máquinas de Estados Finitas

- **Máquinas de Estados Finitas possuem:**
  - **Um conjunto de estados;**
  - **Uma função de cálculo do próximo estado;**
  - **Uma função de cálculo de saídas.**
- **As duas últimas são determinadas pelo estado atual e pelas entradas**



- **Usaremos aqui uma máquina de Moore (saída depende diretamente apenas do estado atual)**

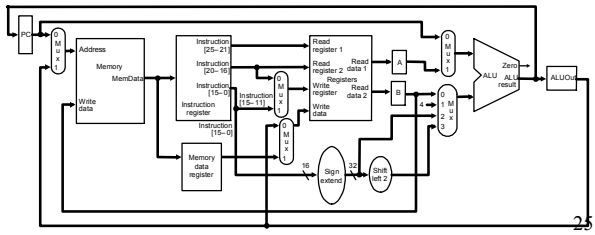
Revisão: Máquinas de Estados Finitas

- **Exemplo:**

B. 21 ***Um amigo gostaria de construir um "olho eletrônico" para simular um dispositivo de segurança. O dispositivo consiste de três luzes em linha, controladas pelas saídas Left, Middle, e Right, que, se ativas, indicam que uma luz deve acender. Somente uma luz acende de cada vez, e a luz "se move" da esquerda para a direita e em seguida da direita para a esquerda, assustando ladrões que acreditam que o dispositivo está monitorando sua atividade. Desenhe uma representação gráfica para a máquina de estados finita usada para especificar o olho eletrônico. Note que a taxa do movimento do olho será controlada pela frequência do relógio do sistema (que não deve ser muito alta) e que não existe essencialmente nenhuma entrada no sistema.***

Abordagem Multiciclo

- Quebra-se as instruções em passos, cada um executando em um ciclo
  - A quantidade de trabalho por ciclo deve ser balanceada
  - Restringe-se cada ciclo a usar apenas uma unidade funcional
- No final de um ciclo
  - Armazena-se valores para uso em ciclos posteriores
  - Introduz-se registradores adicionais "internos"



Exemplo: Execução em Cinco Passos

- Busca de Instrução
- Decodificação de Instrução e Busca de Operandos
- Execução, Cálculo de Endereço de Memória, ou Finalização de um salto
- Acesso à Memória ou Finalização de (Algumas) Instruções tipo R
- Passo de Escrita no Banco de Registradores (*Write-back*)

INSTRUÇÕES LEVAM DE 3 A 5 CICLOS!

Step 1: Instruction Fetch

- Use PC to get instruction and put it in the Instruction Register.
- Increment the PC by 4 and put the result back in the PC.
- Can be described succinctly using RTL "Register-Transfer Language"

```
IR = Memory[PC];
PC = PC + 4;
```

Can we figure out the values of the control signals?

What is the advantage of updating the PC now?

Step 2: Instruction Decode and Register Fetch

- Read registers rs and rt in case we need them
- Compute the branch address in case the instruction is a branch
- RTL:

```
A = Reg[IR[25-21]];
B = Reg[IR[20-16]];
ALUOut = PC + (sign-extend(IR[15-0]) << 2);
```

- We aren't setting any control lines based on the instruction type
  (we are busy "decoding" it in our control logic)

Step 3 (instruction dependent)

- **ALU is performing one of three functions, based on instruction type**
- **Memory Reference:**

```
ALUOut = A + sign-extend(IR[15-0]);
```

- **R-type:**

```
ALUOut = A op B;
```

- **Branch:**

```
if (A==B) PC = ALUOut;
```


Step 4 (R-type or memory-access)

- **Loads and stores access memory**

```
MDR = Memory[ALUOut];
      or
Memory[ALUOut] = B;
```

- **R-type instructions finish**

```
Reg[IR[15-11]] = ALUOut;
```

The write actually takes place at the end of the cycle on the edge

Write-back step

- `Reg[IR[20-16]]= MDR;`

What about all the other instructions?

Summary:

Step name	Action for R-type instructions	Action for memory-reference instructions	Action for branches	Action for jumps
Instruction fetch	IR = Memory[PC] PC = PC + 4			
Instruction decode/register fetch	A = Reg [IR[25-21]] B = Reg [IR[20-16]] ALUOut = PC + (sign-extend (IR[15-0]) << 2)			
Execution, address computation, branch/ jump completion	ALUOut = A op B	ALUOut = A + sign-extend (IR[15-0])	if (A ==B) then PC = ALUOut	PC = PC [31-28] II (IR[25-0]<<2)
Memory access or R-type completion	Reg [IR[15-11]] = ALUOut	Load: MDR = Memory[ALUOut] or Store: Memory [ALUOut] = B		
Memory read completion		Load: Reg[IR[20-16]] = MDR		

Simple Questions

- How many cycles will it take to execute this code?

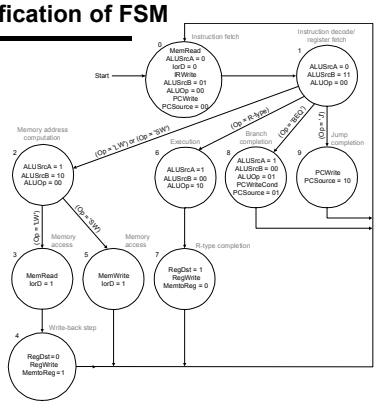
```
            lw $t2, 0($t3)
            lw $t3, 4($t3)
            beq $t2, $t3, Label      #assume not
            add $t5, $t2, $t3
            sw $t5, 8($t3)
Label:      ...
```

- What is going on during the 8th cycle of execution?
- In what cycle does the actual addition of `$t2` and `$t3` takes place?

Implementing the Control

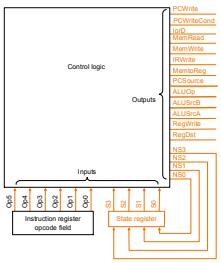
- Value of control signals is dependent upon:
  - what instruction is being executed
  - which step is being performed
- Use the information we've acculumated to specify a finite state machine
  - specify the finite state machine graphically, or
  - use microprogramming
- Implementation can be derived from specification

Graphical Specification of FSM



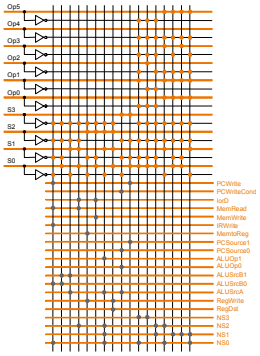
Finite State Machine for Control

- Implementation:



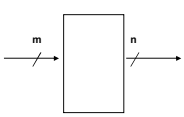
PLA Implementation

- If I picked a horizontal or vertical line could you explain it?



ROM Implementation

- ROM = "Read Only Memory"
  - values of memory locations are fixed ahead of time
- A ROM can be used to implement a truth table
  - if the address is m-bits, we can address 2^m entries in the ROM.
  - our outputs are the bits of data that the address points to.



0	0	0	0	0	1	1
0	0	1	1	1	0	0
0	1	0	1	1	0	0
0	1	1	1	1	0	0
1	0	0	0	0	0	0
1	0	1	0	0	0	1
1	1	0	0	1	1	0
1	1	1	0	1	1	1

m is the "heigth", and n is the "width"

ROM Implementation

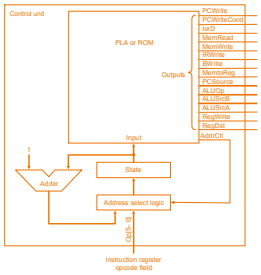
- How many inputs are there?
  6 bits for opcode, 4 bits for state = 10 address lines
  (i.e., 2^{10} = 1024 different addresses)
- How many outputs are there?
  16 datapath-control outputs, 4 state bits = 20 outputs
- ROM is 2^{10} x 20 = 20K bits (and a rather unusual size)
- Rather wasteful, since for lots of the entries, the outputs are the same
  — i.e., opcode is often ignored

ROM vs PLA

- Break up the table into two parts
  — 4 state bits tell you the 16 outputs, 2^4 x 16 bits of ROM
  — 10 bits tell you the 4 next state bits, 2^{10} x 4 bits of ROM
  — Total: 4.3K bits of ROM
- PLA is much smaller
  — can share product terms
  — only need entries that produce an active output
  — can take into account don't cares
- Size is (#inputs × #product-terms) + (#outputs × #product-terms)
  For this example = (10x17)+(20x17) = 460 PLA cells
- PLA cells usually about the size of a ROM cell (slightly bigger)

Another Implementation Style

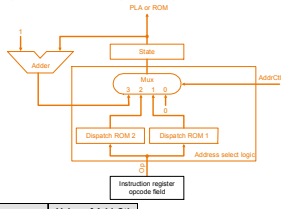
- **Complex instructions: the "next state" is often current state + 1**



Details

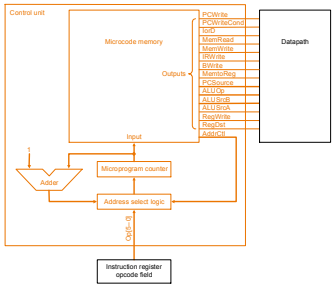
Dispatch ROM 1		
Op	Opcode name	Value
000000	R-format	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

Dispatch ROM 2		
Op	Opcode name	Value
100011	lw	0011
101011	sw	0101



State number	Address-control action	Value of AddrCtl
0	Use incremented state	3
1	Use dispatch ROM 1	1
2	Use dispatch ROM 2	2
3	Use incremented state	3
4	Replace state number by 0	0
5	Replace state number by 0	0
6	Use incremented state	3
7	Replace state number by 0	0
8	Replace state number by 0	0
9	Replace state number by 0	0

Microprogramming



- **What are the "microinstructions" ?**

Microprogramming

- **A specification methodology**
  - **appropriate if hundreds of opcodes, modes, cycles, etc.**
  - **signals specified symbolically using microinstructions**

Label	ALU control	SRC1	SRC2	Register control	Memory	PCWrite control	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	Extshft	Read			Dispatch 1
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch
BEQ1	Subt	A	B			ALUOut-cond	Fetch
JUMP1						Jump address	Fetch

- ***Will two implementations of the same architecture have the same microcode?***
- ***What would a microassembler do?***

Microinstruction format

Field name	Value	Signals active	Comment
ALU control	Add	ALUOp = 00	Cause the ALU to add.
	Subt	ALUOp = 01	Cause the ALU to subtract; this implements the compare for branches.
	Func code	ALUOp = 10	Use the instruction's function code to determine ALU control.
SRC1	PC	ALUSrcA = 0	Use the PC as the first ALU input.
	A	ALUSrcA = 1	Register A is the first ALU input.
SRC2	B	ALUSrcB = 00	Register B is the second ALU input.
	4	ALUSrcB = 01	Use 4 as the second ALU input.
	Extend	ALUSrcB = 10	Use output of the sign extension unit as the second ALU input.
	Extshft	ALUSrcB = 11	Use the output of the shift-by-two unit as the second ALU input.
Register control	Read		Read two registers using the rs and rt fields of the IR as the register numbers and putting the data into registers A and B.
	Write ALU	RegWrite, RegDst = 1, MemtoReg = 0	Write a register using the rd field of the IR as the register number and the contents of the ALUOut as the data.
	Write MDR	RegWrite, RegDst = 0, MemtoReg = 1	Write a register using the rt field of the IR as the register number and the contents of the MDR as the data.
Memory	Read PC	MemRead, IorD = 0	Read memory using the PC as address; write result into IR (and the MDR).
	Read ALU	MemRead, IorD = 1	Read memory using the ALUOut as address; write result into MDR.
	Write ALU	MemWrite, IorD = 1	Write memory using the ALUOut as address, contents of B as the data.
PC write control	ALU	PCSource = 00 PCWrite	Write the output of the ALU into the PC.
	ALUOut-cond	PCSource = 01, PCWriteCond	If the Zero output of the ALU is active, write the PC with the contents of the register ALUOut.
	jump address	PCSource = 10, PCWrite	Write the PC with the jump address from the instruction.
Sequencing	Seq	AddrCtl = 11	Choose the next microinstruction sequentially.
	Fetch	AddrCtl = 00	Go to the first microinstruction to begin a new instruction.
	Dispatch 1	AddrCtl = 01	Dispatch using the ROM 1.
	Dispatch 2	AddrCtl = 10	Dispatch using the ROM 2.

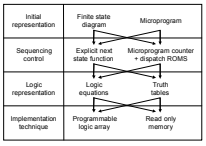
Maximally vs. Minimally Encoded

- **No encoding:**
  - **1 bit for each datapath operation**
  - **faster, requires more memory (logic)**
  - **used for Vax 780 — an astonishing 400K of memory!**
- **Lots of encoding:**
  - **send the microinstructions through logic to get control signals**
  - **uses less memory, slower**
- **Historical context of CISC:**
  - **Too much logic to put on a single chip with everything else**
  - **Use a ROM (or even RAM) to hold the microcode**
  - **It's easy to add new instructions**

Microcode: Trade-offs

- **Distinction between specification and implementation is sometimes blurred**
- **Specification Advantages:**
  - **Easy to design and write**
  - **Design architecture and microcode in parallel**
- **Implementation (off-chip ROM) Advantages**
  - **Easy to change since values are in memory**
  - **Can emulate other architectures**
  - **Can make use of internal registers**
- **Implementation Disadvantages, SLOWER now that:**
  - **Control is implemented on same chip as processor**
  - **ROM is no longer faster than RAM**
  - **No need to go back and make changes**

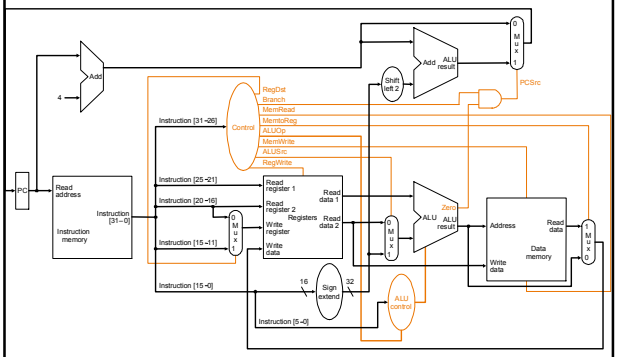
The Big Picture



Exercícios

- **Na Implementação Multiciclo da próxima lâmina, diga:**
  - **5.1: Que instruções, se existe alguma, não continuariam funcionando se:**
    - RegDst = 0
    - ALUSrc = 0
    - MemtoReg = 0
    - Zero = 0
  - **5.2: Que instruções, se existe alguma, não continuariam funcionando se:**
    - RegDst = 1
    - ALUSrc = 1
    - MemtoReg = 1
    - Zero = 1

Uma Implementação Monociclo



Exercícios

- **Na Implementação Multiciclo da próxima lâmina, diga:**
  - **5.3: Que instruções, se existe alguma, não continuariam funcionando se:**
    - RegDst = 0
    - MemtoReg = 0 e IorD = 0
    - AluSrcA = 0
  - **5.4: Que instruções, se existe alguma, não continuariam funcionando se:**
    - RegDst = 1
    - MemtoReg = 1
    - IorD = 1
    - AluSrcA = 1

Uma Implementação Multiciclo