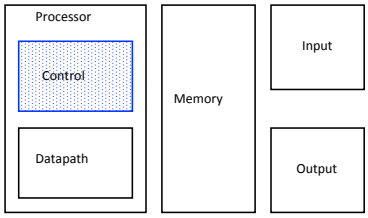
O Processador (CPU)
Implementação *Single-Cycle*
Construção da Unidade de Controle

António de Brito Ferrari
ferrari@ua.pt

Definição da Unidade de Controle

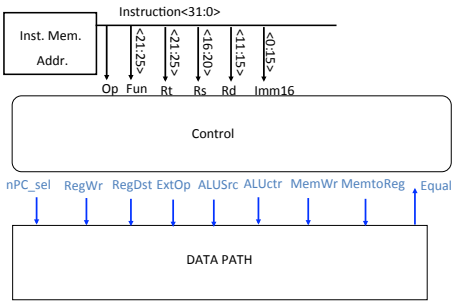


Fases da conceção de um processador

1. Analisar o instruction set => datapath requirements
   - O significado de cada instrução é dado pelas *transferências entre registos*
   - datapath tem de incluir hardware para os registos do ISA
   - datapath tem de suportar as transferências entre registos
2. Selecionar os componentes para o datapath e definir a metodologia para os impulsos de relógio
3. Construir o datapath de modo a satisfazer as especificações
4. **Analisar a implementação de cada instrução para identificar os sinais de controlo que acionam as transferências entre registos**
5. Realizar a lógica de controlo

4. Identificar os sinais de controlo que acionam as transferências entre registos

Datapath: RTL -> Control



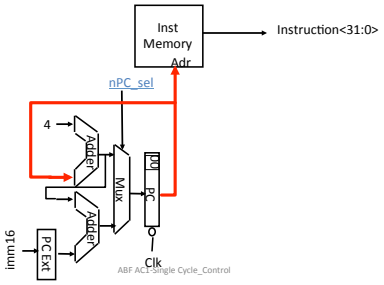
RTL: a instrução **Add**

31–26	25–21	20–16	15–11	10–6	5–0
op	rs	rt	rd	shamt	funct
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

- **addrd, rs, rt**
  - mem[PC] Fetch da instrução da memória
  - R[rd] <- R[rs] + R[rt] Operação especificada
  - PC <- PC + 4 Calcular o endereço da instrução seguinte

Instruction Fetch Unit no início de Add

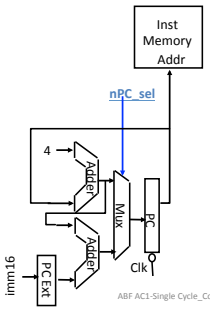
- Fetch da instrução da Instruction Memory: Instruction <- mem[PC]



Significado dos Sinais de Control

Rs, Rt, Rd e Imed16 ligados diretamente ao datapath

nPC_sel: 0 => PC <– PC + 4; 1 => PC <– PC + 4 + SignExt(Im16)



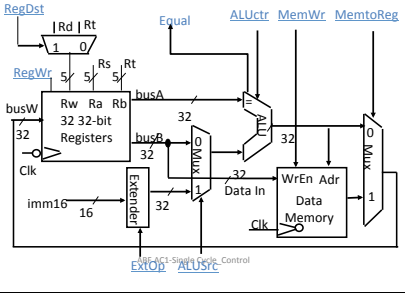
Significado dos Sinais de Control (2)

ExtOp: "zero", "sign"

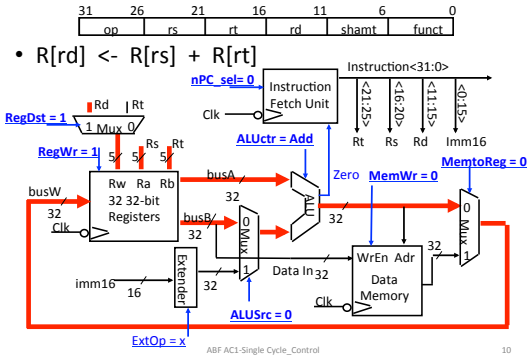
ALUsrc: **0** => regB; **1** => immed

ALUctr: "add", "sub", "or"

- MemWr: write memory
- MemtoReg: **1** => Mem
- RegDst: **0** => "rt"; **1** => "rd"
- RegWr: write dest register

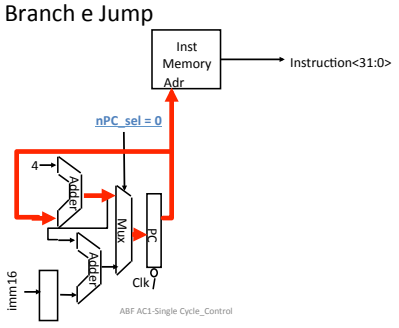


O Datapath durante *Add*

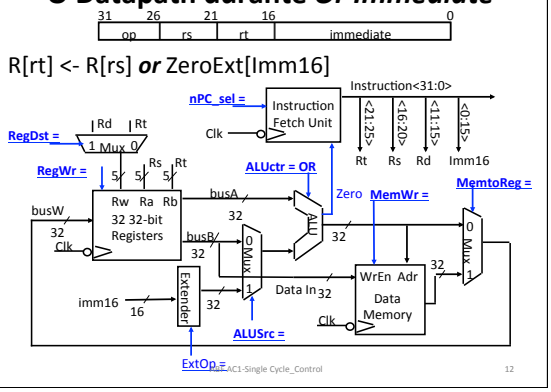


Instruction Fetch Unit no fim de *Add*

PC <- PC + 4 # Igual para todas as instruções exceto Branch e Jump



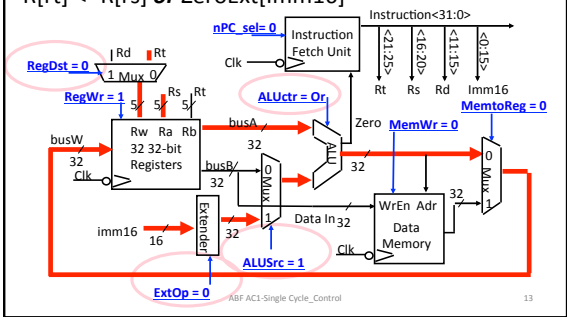
O Datapath durante *Or Immediate*



O Datapath durante *Or Immediate*

31 26 21 16 0
op rs rt immediate

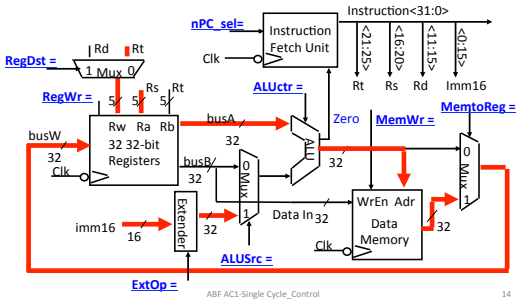
R[rt] <- R[rs] ***or*** ZeroExt[Imm16]



O Datapath durante *Load*

31 26 21 16 0
op rs rt immediate

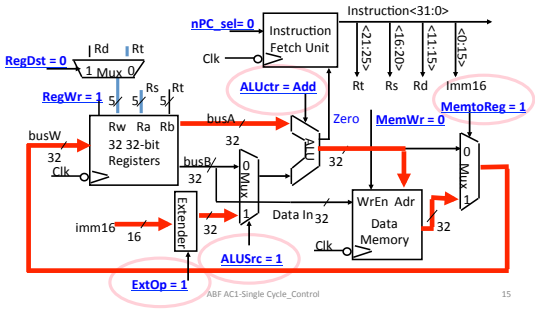
R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



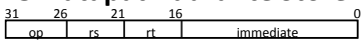
O Datapath durante *Load*

31 26 21 16 0
op rs rt immediate

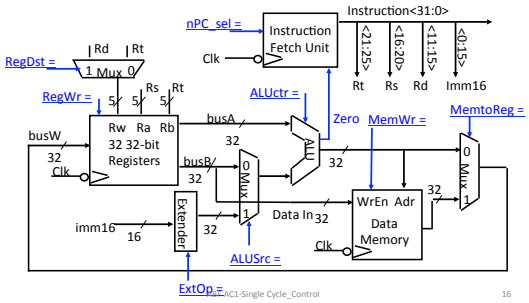
R[rt] <- Data Memory {R[rs] + SignExt[imm16]}



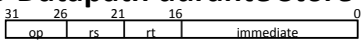
O Datapath durante *Store*



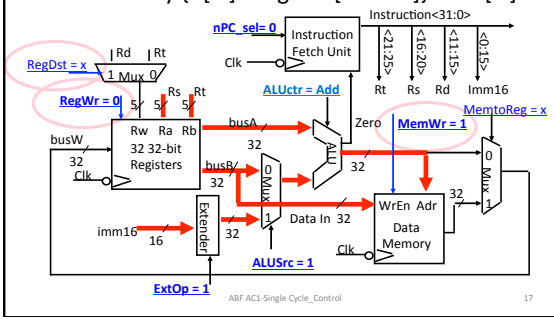
Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



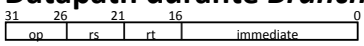
O Datapath durante *Store*



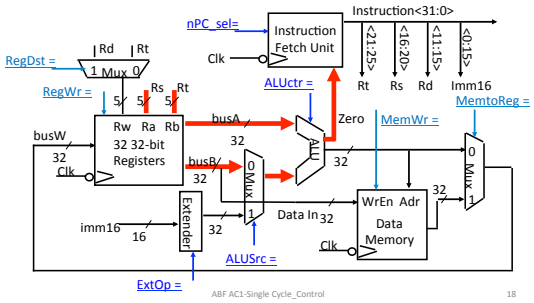
Data Memory {R[rs] + SignExt[imm16]} <- R[rt]



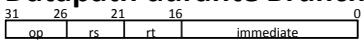
O Datapath durante *Branch*



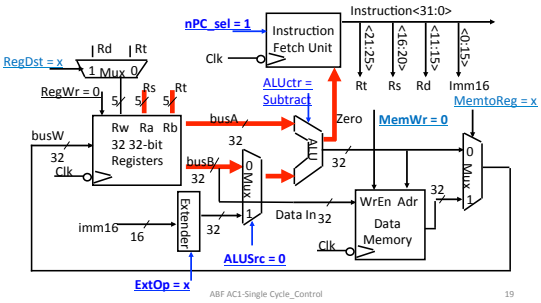
if (R[rs] - R[rt] == 0) then Zero <- 1 ; else Zero <- 0



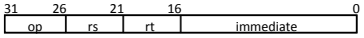
O Datapath durante *Branch*



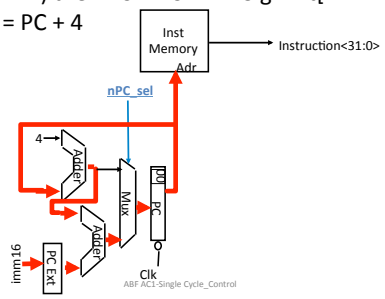
if (R[rs] - R[rt] == 0) then Zero <- 1; else Zero <- 0



Instr. Fetch Unit no fim de Branch



if (Zero == 1) then PC = PC + 4 + SignExt[imm16]*4;
else PC = PC + 4



ALU
Definição da Operação a Executar

ALUctr	Function
0000	AND
0001	OR
0010	ADD
0110	SUB
0111	Set on Less Than
1100	NOR

Sinais de Control

inst	Register Transfer	
ADD	R[rd] <– R[rs] + R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = "add", RegDst = rd, RegWr, nPC_sel = "+4"	
SUB	R[rd] <– R[rs] – R[rt];	PC <– PC + 4
	ALUsrc = RegB, ALUctr = "sub", RegDst = rd, RegWr, nPC_sel = "+4"	
ORi	R[rt] <– R[rs] + zero_ext(Imm16);	PC <– PC + 4
	ALUsrc = Im, Extop = "Z", ALUctr = "or", RegDst = rt, RegWr, nPC_sel = "+4"	
LOAD	R[rt] <– MEM[R[rs] + sign_ext(Imm16)];	PC <– PC + 4
	ALUsrc = Im, Extop = "Sn", ALUctr = "add", MemtoReg, RegDst = rt, RegWr, nPC_sel = "+4"	
STORE	MEM[R[rs] + sign_ext(Imm16)] <– R[rs];	PC <– PC + 4
	ALUsrc = Im, Extop = "Sn", ALUctr = "add", MemWr, nPC_sel = "+4"	
BEQ	if (R[rs] == R[rt]) then PC <– PC + sign_ext(Imm16)] || 00 else PC <– PC + 4	
	nPC_sel = EQUAL, ALUctr = "sub"	

5. Realização da lógica de controlo

Fases da conceção de um processador

1. Analisar o instruction set => datapath requirements
   - O significado de cada instrução é dado pelas *transferências entre registos*
   - datapath tem de incluir hardware para os registos do ISA
   - datapath tem de suportar as transferências entre registos
2. Selecionar os componentes para o datapath e definir a metodologia para os impulsos de relógio
3. Construir o datapath de modo a satisfazer as especificações
4. Analisar a implementação de cada instrução para identificar os sinais de controlo que acionam as transferências entre registos
5. **Realizar a lógica de controlo**

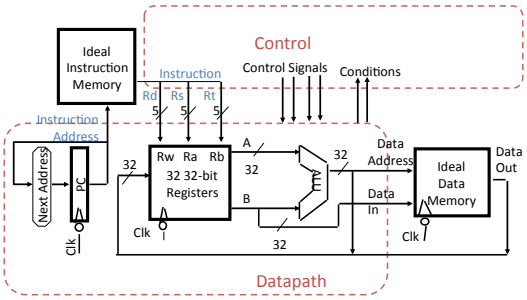
Lógica para cada sinal de control (1/2)

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == "000000" **OR** OP == "000100") then "regB"
  else "immed"
- ALUctr <= if (OP == "000000") then funct
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"
- ExtOp <= _____
- MemWr <= _____
- MemtoReg <= _____
- RegWr: <=_____
- RegDst: <= _____

Lógica para cada sinal de control (2/2)

- nPC_sel <= if (OP == BEQ) then EQUAL else 0
- ALUsrc <= if (OP == "000000") then "regB" else "immed"
- ALUctr <= if (OP == "000000") then funct
  elseif (OP == ORi) then "OR"
  elseif (OP == BEQ) then "sub"
  else "add"
- ExtOp <= if (OP == ORi) then "zero" else "sign"
- MemWr <= (OP == Store)
- MemtoReg <= (OP == Load)
- RegWr: <= if ((OP == Store) || (OP == BEQ)) then **0** else **1**
- RegDst: <= if ((OP == Load) || (OP == ORi)) then **0** else **1**

Uma visão da Implementação



- Logical vs. Physical Structure

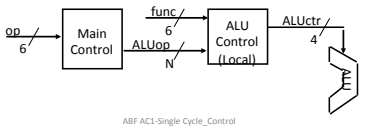
Sumário dos Sinais de Control

func	10 0000	10 0010	Don't Care				
op	00 0000	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	add	sub	ori	lw	sw	beq	jump
RegDst	1	1	0	0	x	x	x
ALUSrc	0	0	1	1	1	0	x
MemtoReg	0	0	0	1	x	x	x
RegWrite	1	1	1	1	0	0	0
MemWrite	0	0	0	0	1	0	0
nPCsel	0	0	0	0	0	1	0
Jump	0	0	0	0	0	0	1
ExtOp	x	x	0	1	1	x	x
ALUctr<3:0>	Add	Subtract	Or	Add	Add	Subtract	xxx

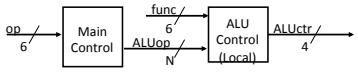
	31–26	21–26	16–21	11–16	6–11	0–6	
R-type	op	rs	rt	rd	shamt	funct	add, sub
I-type	op	rs	rt	immediate			ori, lw, sw, beq
J-type	op	target address					jump

O Conceito de Descodificação Local

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop<N:0>	"R-type"	Or	Add	Add	Subtract	xxx



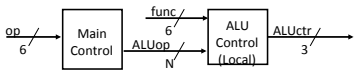
Codificação de ALUop



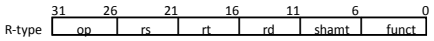
- ALUop têm de ter 3 bits para representar:
  - (1) "R-type" instructions
  - "I-type" instructions que requerem que a ALU execute:
    - (2) Or, (3) Add, e (4) Subtract

	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	"R-type"	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx

A Descodificação de “funct”



	R-type	ori	lw	sw	beq	jump
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01	xxx



funct<5:0>	Instruction Operation
10 0000	add
10 0010	subtract
10 0100	and
10 0101	or
10 1010	set-on-less-than

ALUctr<3:0>	ALU Operation
0010	Add
0110	Subtract
0000	And
0001	Or
0111	Set-on-less-than

ALUctr Truth Table

ALUop (Symbolic)	R-type	ori	lw	sw	beq
	“R-type”	Or	Add	Add	Subtract
ALUop<2:0>	1 00	0 10	0 00	0 00	0 01

funct<3:0>	Instruction Op.
0000	add
0010	subtract
0100	and
0101	or
1010	set-on-less-than

ALUop			func				ALU Operation	ALUctr		
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>		bit<2>	bit<1>	bit<0>
0	0	0	x	x	x	x	Add	0	1	0
0	x	1	x	x	x	x	Subtract	1	1	0
0	1	x	x	x	x	x	Or	0	0	1
1	x	x	0	0	0	0	Add	0	1	0
1	x	x	0	0	1	0	Subtract	1	1	0
1	x	x	0	1	0	0	And	0	0	0
1	x	x	0	1	0	1	Or	0	0	1
1	x	x	1	0	1	0	Set on <	1	1	1

Equação Lógica de ALUctr<2>

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<2>
0	x	1	x	x	x	x	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

func<3> don’t care

ALUctr<2> = !ALUop<2> & ALUop<0> +

ALUop<2> & !func<2> & func<1> & !func<0>

Equação Lógica de ALUctr<1>

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<1>
0	0	0	x	x	x	x	1
0	x	1	x	x	x	x	1
1	x	x	0	0	0	0	1
1	x	x	0	0	1	0	1
1	x	x	1	0	1	0	1

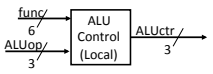
- ALUctr<1> = !ALUop<2> & !ALUop<0> +
  ALUop<2> & !func<2> & !func<0>

Equação Lógica de ALUctr<0>

ALUop			func				
bit<2>	bit<1>	bit<0>	bit<3>	bit<2>	bit<1>	bit<0>	ALUctr<0>
0	1	x	x	x	x	x	1
1	x	x	0	1	0	1	1
1	x	x	1	0	1	0	1

ALUctr<0> = !ALUop<2> & ALUop<1>
+ ALUop<2> & !func<3> & !func<2> & !func<0>
+ ALUop<2> & func<3> & !func<2> & func<1>
& !func<0>

ALU Control Block



ALUctr<2> = !ALUop<2> & ALUop<0> +
ALUop<2> & !func<2> & func<1> & !func<0>
ALUctr<1> = !ALUop<2> & !ALUop<0> +
ALUop<2> & !func<2> & !func<0>
ALUctr<0> = !ALUop<2> & ALUop<0> +
ALUop<2> & !func<3> & func<2> & !func<1> & func<0>
+ ALUop<2> & func<3> & !func<2> & func<1> & !func<0>

Lógica para os sinais de control

nPC_sel <= if (OP == BEQ) then EQUAL else 0

ALUsrc <= if (OP == “Rtype” OR OP == “BEQ”) then “regB”
else “immed”

ALUctr <= if (OP == “Rtype”) then funct
elseif (OP == ORi) then “OR”
elseif (OP == BEQ) then “sub”
else “add”

ExtOp <= if (OP == ORi) then “zero” else “sign”

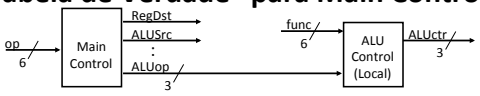
MemWr <= (OP == Store)

MemtoReg <= (OP == Load)

RegWr <= if ((OP == Store)||(OP == BEQ)) then 0 else 1

RegDst <= if ((OP == Load)||(OP == ORi)) then 0 else 1

“Tabela de Verdade” para Main Control



op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegDst	1	0	0	x	x	x
ALUSrc	0	1	1	1	0	x
MemtoReg	0	0	1	x	x	x
RegWrite	1	1	1	0	0	0
MemWrite	0	0	0	1	0	0
Branch	0	0	0	0	1	0
Jump	0	0	0	0	0	1
ExtOp	x	0	1	1	x	x
ALUop (Symbolic)	“R-type”	Or	Add	Add	Subtract	xxx
ALUop <2>	1	0	0	0	0	x
ALUop <1>	0	1	0	0	0	x
ALUop <0>	0	0	0	0	1	x

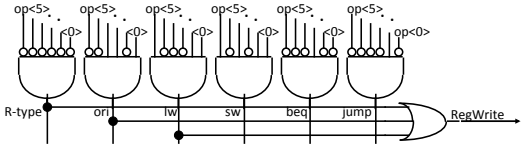
“Tabela de Verdade” para RegWrite

op	00 0000	00 1101	10 0011	10 1011	00 0100	00 0010
	R-type	ori	lw	sw	beq	jump
RegWrite	1	1	1	0	0	0

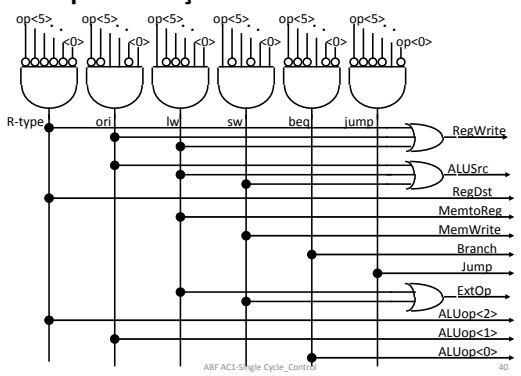
RegWrite = R-type + ori + lw

= !op<5> & !op<4> & !op<3> & !op<2> & !op<1> & !op<0> (R-type)
+ !op<5> & !op<4> & op<3> & op<2> & !op<1> & op<0> (ori)

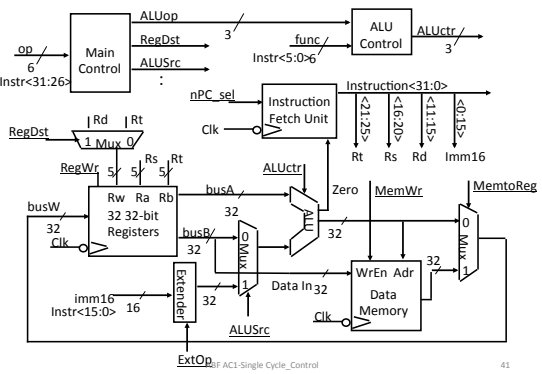
+ op<5> & !op<4> & !op<3> & !op<2> & op<1> & op<0> (lw)



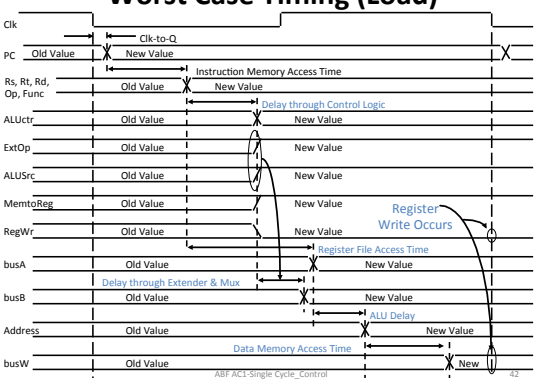
Implementação PLA de Main Control



Single Cycle Processor: esquema global



Worst Case Timing (Load)



Desvantagens de Single Cycle CPU

- Long cycle time – suficientemente longo para a execução de ***load***:
  - PC's Clock -to-Q +
  - Instruction Memory Access Time +
  - Register File Access Time +
  - ALU Delay (address calculation) +
  - Data Memory Access Time +
  - Register File Setup Time +
  - Clock Skew
- **Cycle time para load muito mais longo do que o necessário para todas as outras instruções**

Sumário (1/2)

- **5 etapas no desenho de um processador**
  1. Analisar o instruction set => requesitos do datapath
  2. Selecionar os componentes do datapath e definir o clock a usar (single phase, negative edge-triggered)
  3. Montar o datapath satisfazendo os requesitos
  4. Analisar a implementação de cada instrução para determinar o conjunto dos pontos de controle que afetam as transferências entre registos.
  5. Realizar a lógica de controle

Sumário (2/2)

- **MIPS facilita o desenho do processador**
  - Instruções de comprimento fixo (32 bits)
  - Registos indicados sempre nas mesmas posições na instrução
  - Immediates sempre com o mesmo tamanho e localização
  - Operações sempre sobre registos e/ou immediates
- **Single cycle datapath => CPI=1**

Cycle Time longo

Desenhar um processador mais rápido:

MULTI-CYCLE CPU