Introdução à Arquitetura de Computadores

Aula26

µArquitetura MIPS Multicycle: II

Unidade de Controlo

Entradas e Saídas

Máquina de Estados (FSM):

Ciclos de *Fetch* e *Decode*

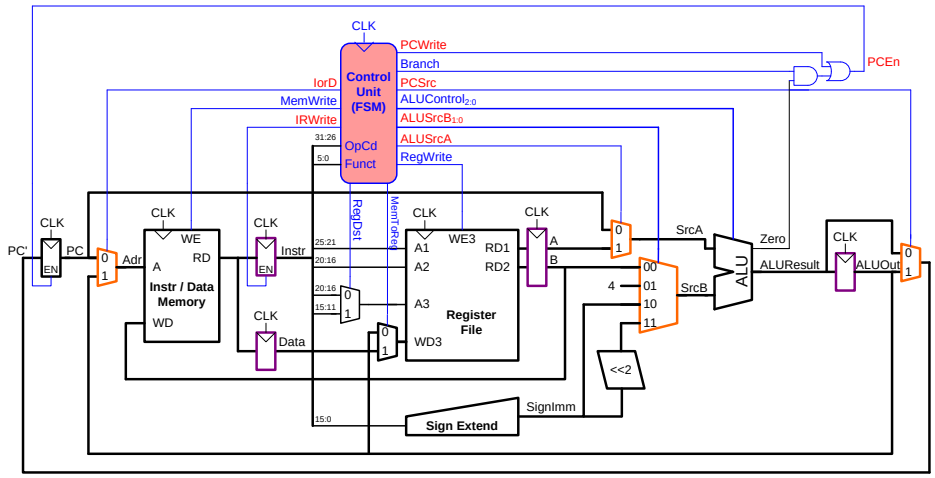
Execução das Instruções:

lw/sw, tipo-R e beq

Extensões

Instruções: addi e j

Datapath MultiCycle - Controlador FSM

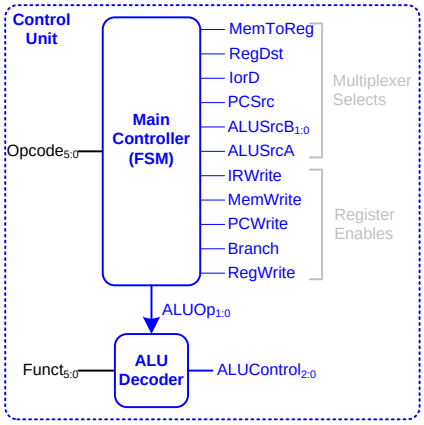


O *Controlador Principal* é uma máquina síncrona (FSM) do tipo Moore responsável pela geração dos sinais de controlo do *datapath multicyle*.

Controlador MC (1) - Principal + ALU Decoder

1. O *Controlador Principal*

é uma máquina síncrona (FSM).



2. O *Descodificador da ALU*

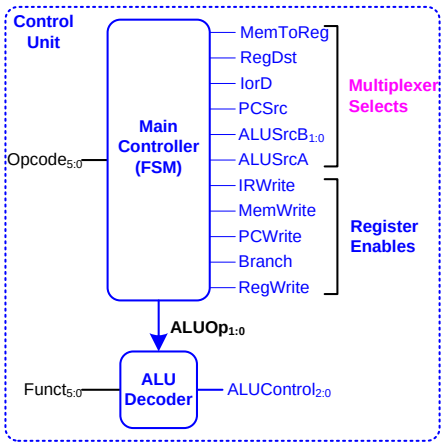
é igual ao do *single-cycle* (lógica combinatória).

$ALUOp_{1:0}$	$Funct_{5:0}$	$ALUControl_{2:0}$
00	X	010 (Add)
01	X	110 (Subtract)
10	100000 (add)	010 (Add)
10	100010 (sub)	110 (Subtract)
10	100100 (and)	000 (And)
10	100101 (or)	001 (Or)
10	101010 (slt)	111 (SLT)

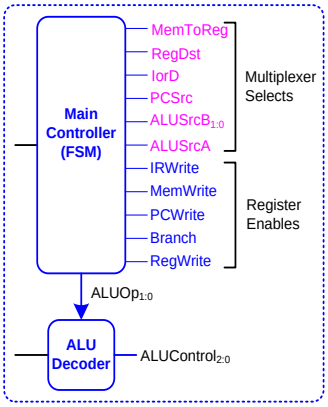
FSM (1) - Tipos de Sinais

O *Controlador Principal* gera 3 tipos de sinais:

- Seleção de *multiplexers*
- *Enable* de registos
- ALUOp



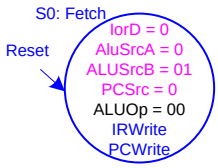
FSM (2) - Estado e Sinais de Controlo



Sinais dentro de cada estado:

1. Seleção de multiplexers: apresentam o respectivo valor binário.
2. *Enable* de registos: só estão indicados quando activos.
3. Valor de ALUOp: não aparece explicitado no *datapath*, sendo substituído pelo ALUControl (gerado pelo ALU Decoder).

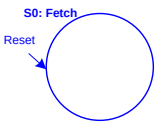
Exemplo:



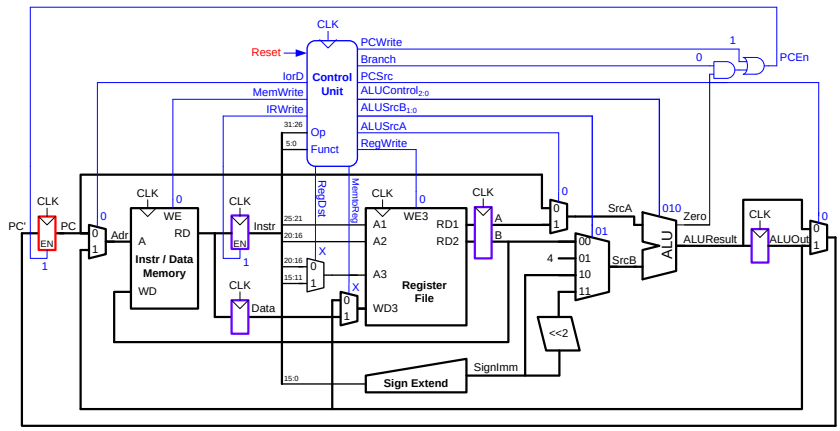
Sobre os diagramas temporais seguintes:

1. Podem ser ignorados numa primeira leitura.
2. Complementam a explicação sobre o funcionamento da FSM da Unidade de Controlo (UC).

FSM (3) - Reset e Fetch: Estado S0



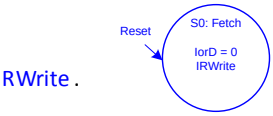
O primeiro passo da execução consiste na operação de leitura da instrução da memória (*fetch*) cujo endereço está contido no registo PC.
A FSM entra no estado de *Fetch* após o sinal de *Reset*.

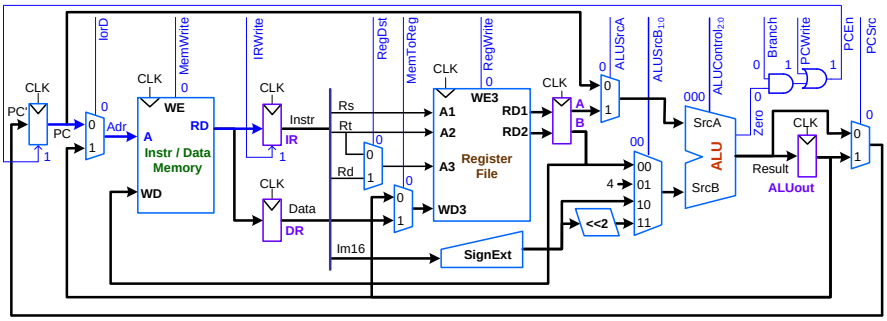


FSM (4) - Fetch (1): Estado S0

S0 - *Fetch:*

Lê a Instrução cujo endereço (Adr) é o valor do PC (IorD=0).
A instrução é escrita no Registo de Instrução, Instr, activando IRWrite.

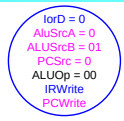


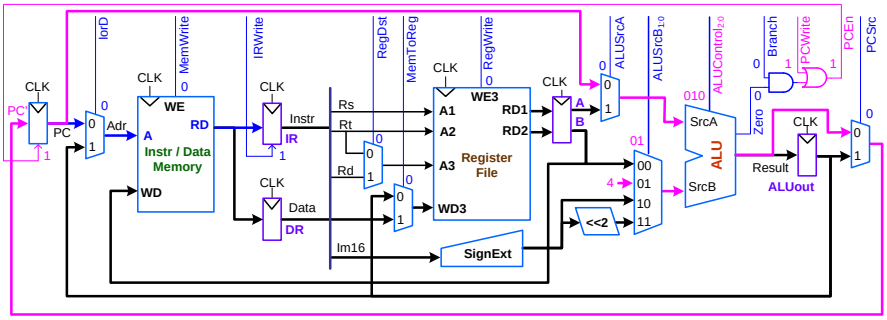


FSM (5) - Fetch e IncPC (2): S0

S0 - *Fetch:*

Lê a Instrução cujo endereço (Adr) é o valor do PC (IorD=0).
A instrução é escrita no Registo de Instrução, Instr, activando IRWrite.

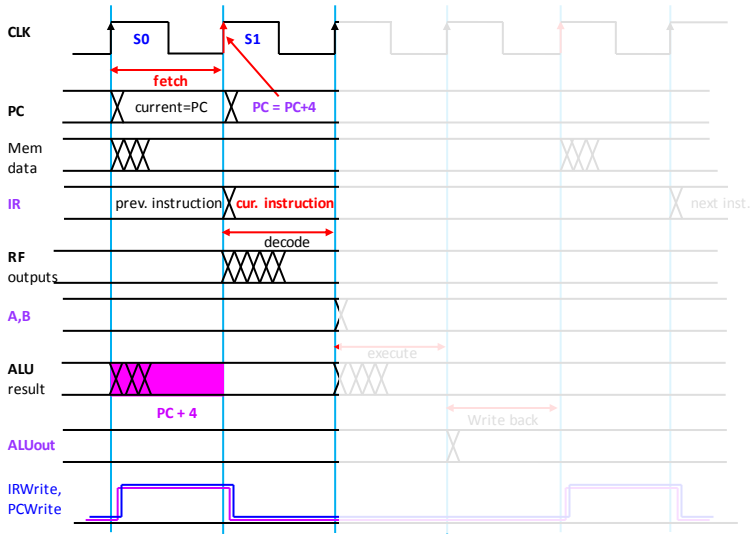




S0 - IncPC: Seleção, ALUOp e Resultado:
ALUSrcA = 0 ->SrcA = PC; ALUSrcB = 01 -> SrcB = 4
ALUOp = 00 ->ALUControl = 010 (*add*); *Result*: PC + 4

Para atualizar o PC com PC': PCSrc = 0 e o PCWrite é activado (-> PCEn=1).

FSM (6) - Fetch e IncPC (3) - Timing

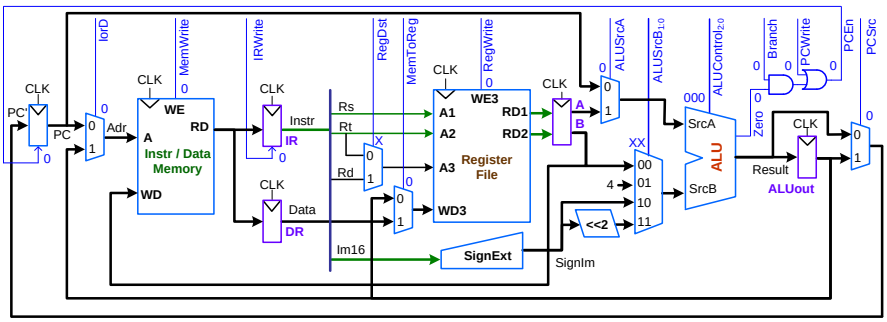


O *fetch* da instrução atual e o cálculo do PC-seguinte são feitos em paralelo.

FSM (7) - Decode: Estado S1

1. O Banco de Registos lê sempre os valores dos dois operandos *Rs* e *Rt* da instrução e coloca-os nos registos A e B respectiva/.

2. Em paralelo, o valor imediato é *sign-extended* de 16 para 32 bits (nas instruções do tipo-R este valor SignIm é ignorado!).



3. A fase de descodificação usa o *opcode* da instrução para decidir o que fazer a seguir.

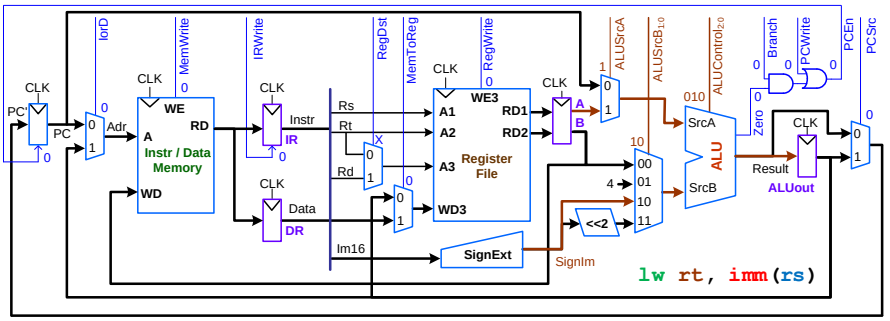
Não são necessários sinais de controlo nesta fase. Todavia, a FSM deve aguardar um ciclo de *clock* para que as operações de leitura e descodificação se completem.

Após S1, os estados seguintes dependem da instrução. Comecemos com lw/sw...

FSM (8) - Cálculo do Endereço: S2

S2: Se for lw ou sw, é calculado o endereço efetivo, adicionando ao Endereço Base (*Rs*->A) o valor imediato (*Im16*) depois de *sign-extended* (*SignIm*). O endereço (*Result*) é guardado em ALUout.

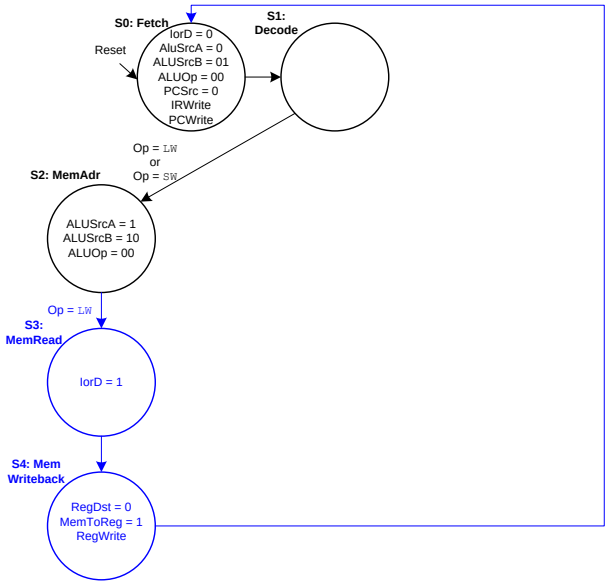
S2: MemAdr
ALUSrcA = 1
ALUSrcB = 10
ALUOp = 00



S2: Seleção, ALUOp e Resultado:

ALUSrcA = 1 -> SrcA = A ; ALUSrcB = 10 -> SrcB = *SignIm*
ALUOp = 00 -> ALUControl = 010 (*add*).
Result = A + *SignIm*

FSM (9) - lw (1): Estados S3 + S4



Leitura do valor da memória e escrita no Banco de Registos.

S3: Para ler da memória, IorD = 1, seleciona o endereço que se encontra em ALUout. O valor lido da memória é armazenado no registo Data .

S4: O valor em Data é escrito no Banco de Registos.
MemToReg = 1 seleciona Data;
RegDst = 0 seleciona o reg. rt ;
RegWrite é activado para escrever no RF, completando a execução de lw.

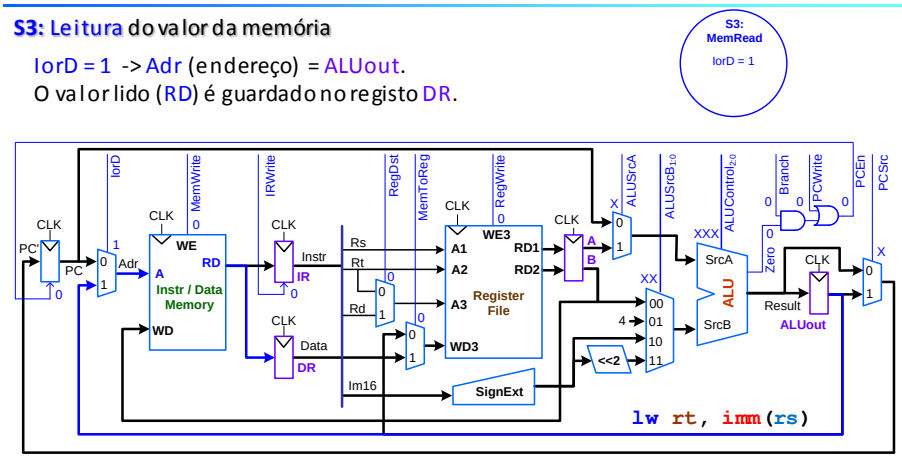
S0: Finalmente, a FSM regressa ao estado inicial, S0, para fazer o *fetch* da instrução seguinte.

lw rt, imm(rs)

FSM (10) - lw (2): S3 - MemRead

S3: Leitura do valor da memória

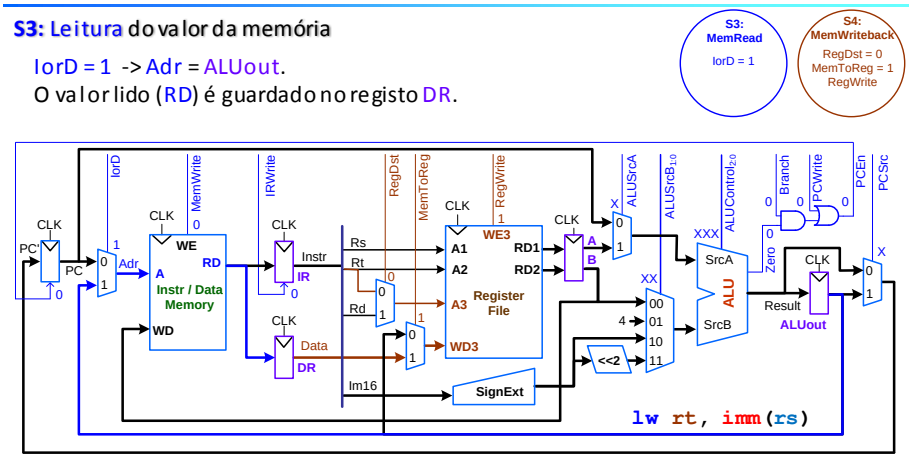
IorD = 1 -> Adr (endereço) = ALUout.
O valor lido (RD) é guardado no registo DR.



FSM (11) - lw (3): S3 + S4 - MemWriteback

S3: Leitura do valor da memória

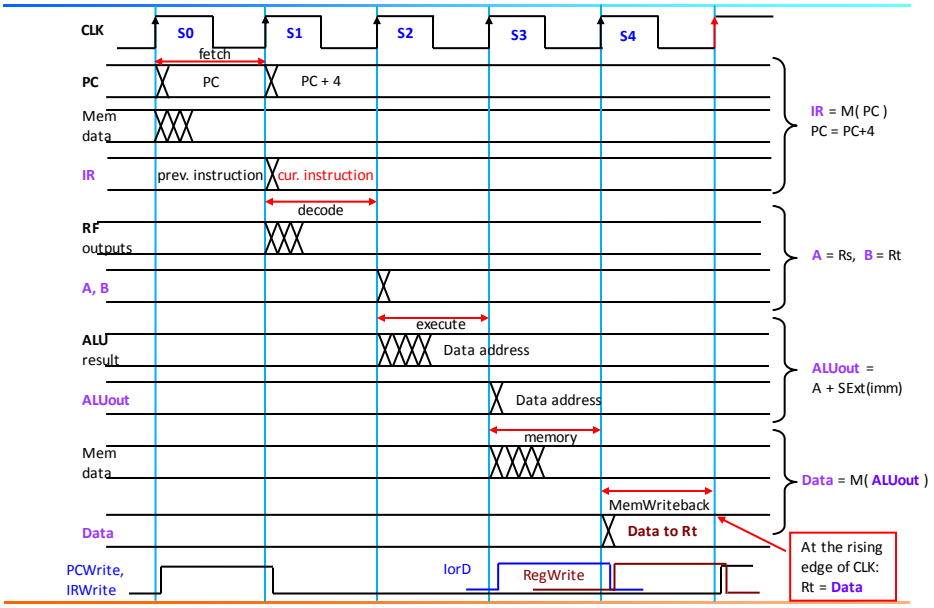
IorD = 1 -> Adr = ALUout.
O valor lido (RD) é guardado no registo DR.



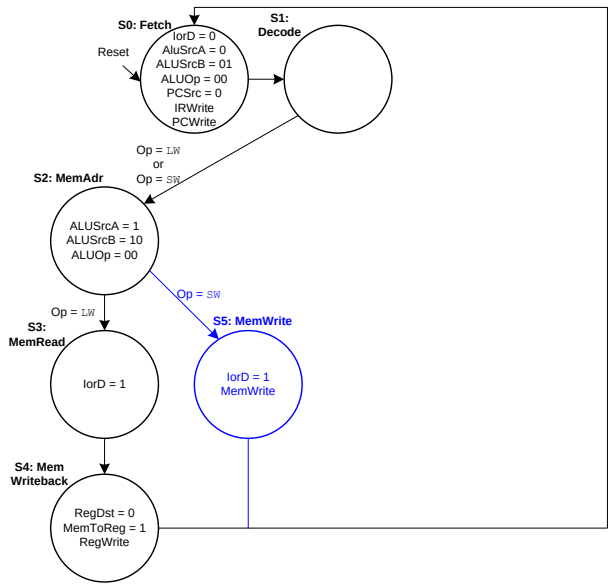
S4: O valor em DR é escrito no Banco de Registos

RegDst = 0 -> A3 = Rt e MemToReg = 1 -> WD3 = Data.
RegWrite é activado para escrever, completando a execução de lw.

FSM (12) - lw (4): Timing - 5 ciclos



FSM (13) - sw (1): Estado S5 - MemWrite



O valor do registo B é escrito na memória.

S5: IorD = 1, seleciona o endereço guardado em ALUout (calculado em S2)

MemWrite é activado para escrever na o valor B, completando a execução de sw.

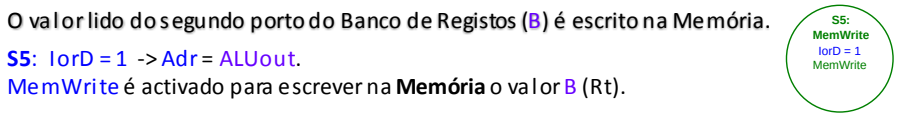
(**menos** um ciclo que lw) .

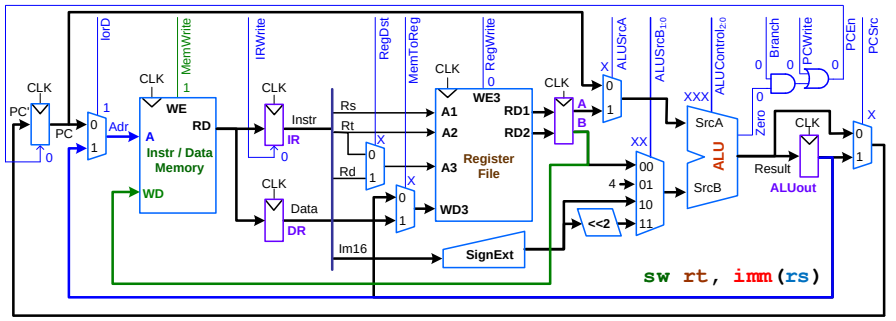
```
sw rt, imm(rs)
```

FSM (14) - sw (2): S5 - MemWrite

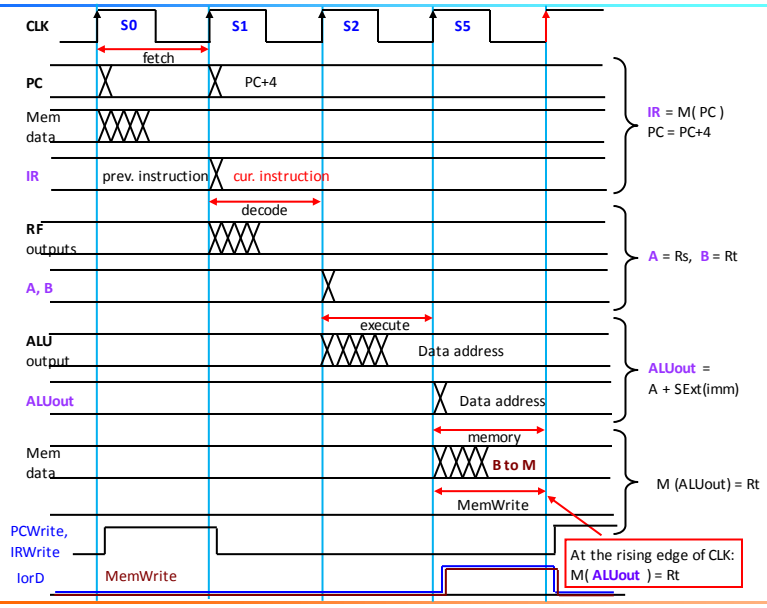
O valor lido do segundo porto do Banco de Registos (B) é escrito na Memória.

S5: IorD = 1 -> Adr = ALUout.
MemWrite é activado para escrever na **Memória** o valor B (Rt).

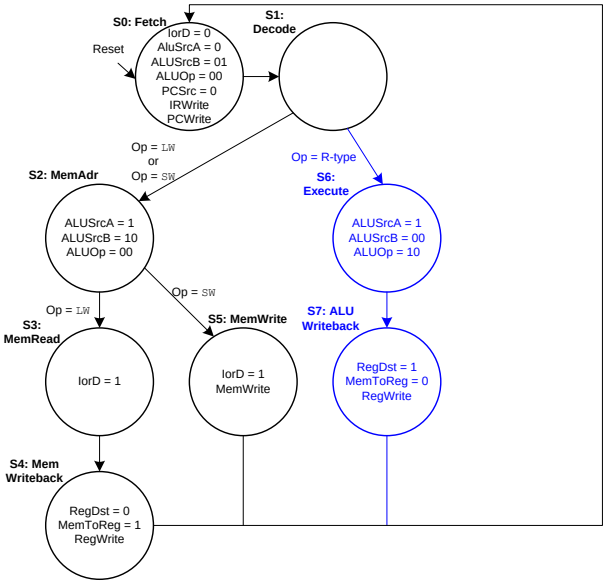




FSM (15) - sw (3): Timing - 4 ciclos



FSM (16) - tipo-R (1): Estados S6 + S7



Calcula o resultado da operação na ALU e escreve-o no Banco de Registos.

S6: São selecionados os registos A e B (ALUSrcA = 1, ALU-SrcB = 00) e calculada a operação indicada pelo campo *funct* da instrução.

O valor de ALUOp é igual a 10 para todas instruções do tipo-R.

O ALUResult é guardado em ALUout.

S7: O valor em ALUout é escrito no Banco de Registos. RegDst = 1, seleciona o registo destino rd.

MemToReg = 0, o **valor** WD3 vem do registo ALUout.

RegWrite é activado para escrever, **completando** a execução.

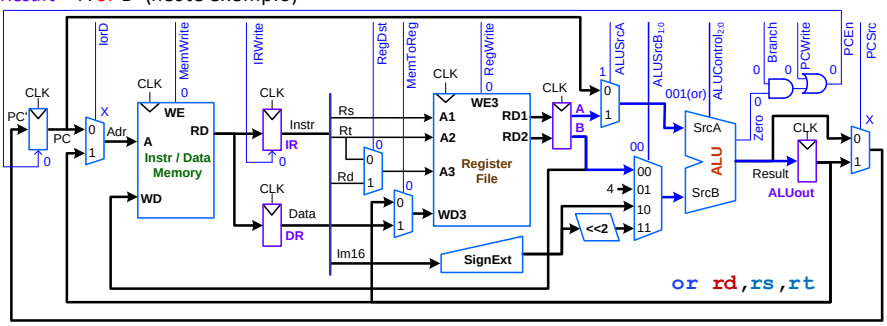
FSM (17) - tipo-R (2): S6 - Execução na ALU

S6: Calcula o resultado da operação na ALU

Seleção e ALUOp: ALUSrcA = 1 -> SrcA = A
ALUSrcB = 00 -> SrcB = B
ALUOp = 10 -> ALUControl = 001 (*or*, ver slide 2).

S6
ALUSrcA = 1
ALUSrcB = 00
ALUOp = 10

Result = A *or* B (neste exemplo)

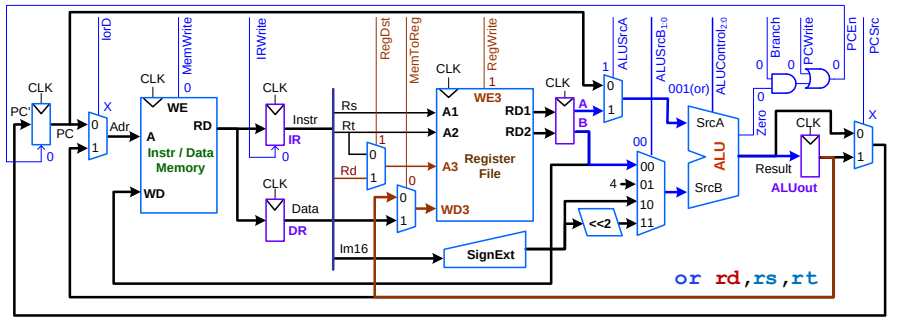


FSM (18) - tipo-R (2): S6 + S7 - ALU Writeback

S6: Calcula o resultado da operação na ALU

Seleção e ALUOp: ALUSrcA = 1 -> SrcA = A
ALUSrcB = 00 -> SrcB = B
ALUOp = 10 -> ALUControl = 001 (*or*).

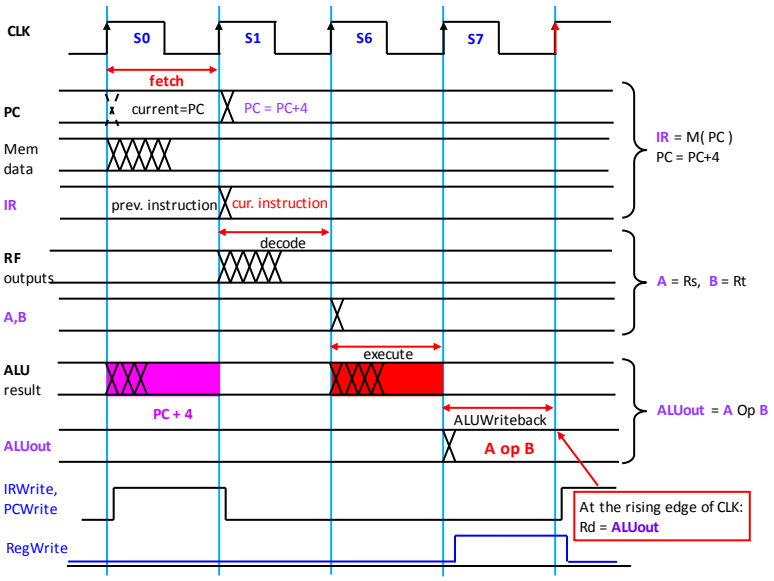
Result = A *or* B



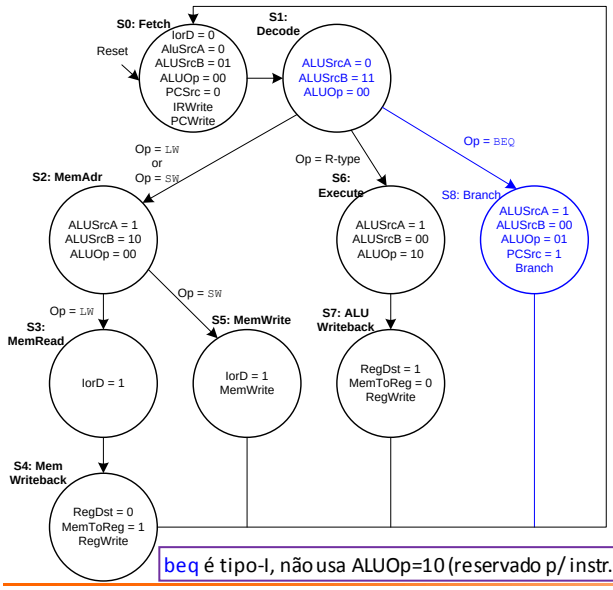
S7: Escrita do valor em ALUout no Banco de Registos

RegDst = 1 -> A3 = Rd e MemToReg = 0 -> WD3 = ALUout.
RegWrite é activado para escrever, completando a execução da instrução.

FSM (19) - tipo-R (3): Timing - 4 ciclos



FSM (20) - beq (1): Estados S1 + S8



O CPU calcula o BTA e compara rs com rt, ambas as operações requerem a ALU

Precisa da ALU mais 2 vezes:
- Uma para calcular o BTA
- Outra para decidir se os valores de rs e rt são iguais

Tira partido do facto da ALU estar livre durante o estado S1 e usa-a para calcular o BTA.

S1: Calcula BTA

BTA = (PC+4) + (SignIm<<2)

S8: Compara o conteúdo dos registos rs e rt. Caso sejam iguais PC' = BTA.

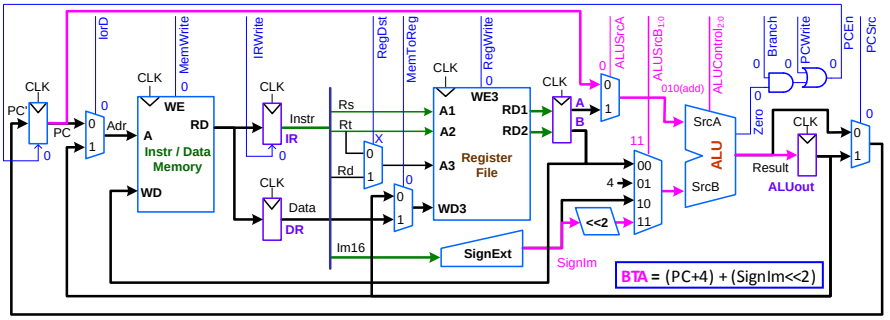
beq é tipo-I, não usa ALUOp=10 (reservado p/ instr. tipo-R).

`beq rs,rt,imm`

FSM (21) - beq (2): S1: BTA (+ Decode)

Após S0, a instrução beq usa a ALU mais **duas** vezes:

S1: Calcula o endereço-alvo (BTA) e guarda-o em ALUout.
Este será, eventualmente, usado em S8.

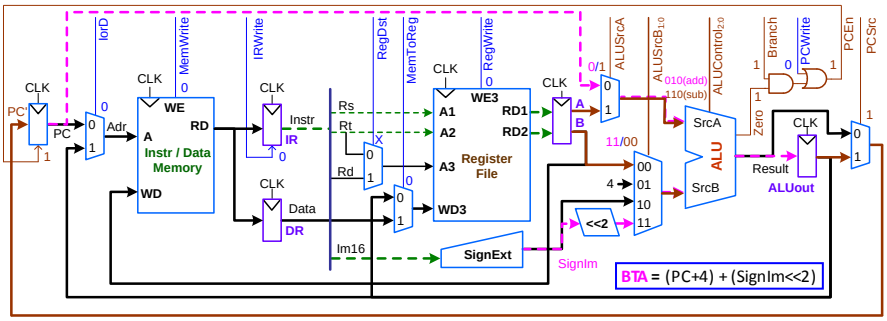


BTA: Seleção, ALUOp e Resultado:
- ALUSrcA = 0 -> SrcA = PC4 (em S1 o valor de PC = PC+4, vide slide 8)
- ALUSrcB = 11 -> SrcB = SignIm<<2
- ALUOp = 00 -> ALUControl = 010 (*add*).
- *Result* = PC4 + SignIm<<2

FSM (22) - beq (3): S1 + S8: Branch

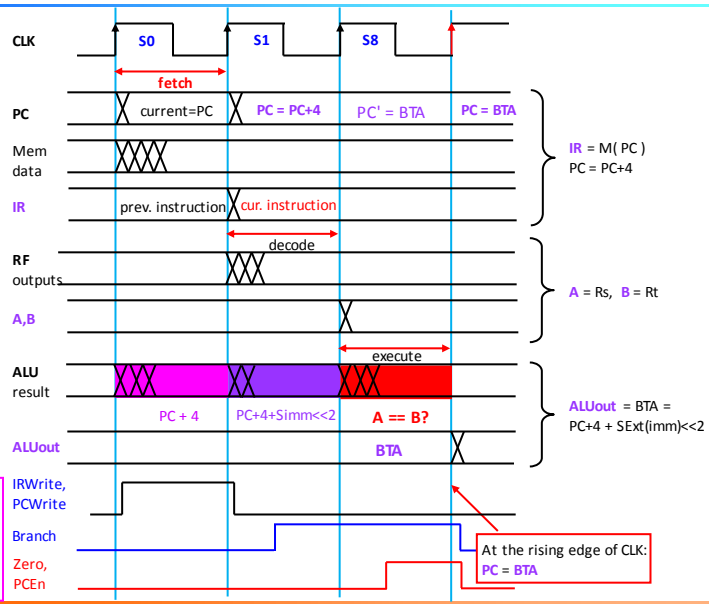
Após S0, a instrução beq usa a ALU mais **duas** vezes:

S1: Calcula o endereço-alvo (BTA) e guarda-o em ALUout.



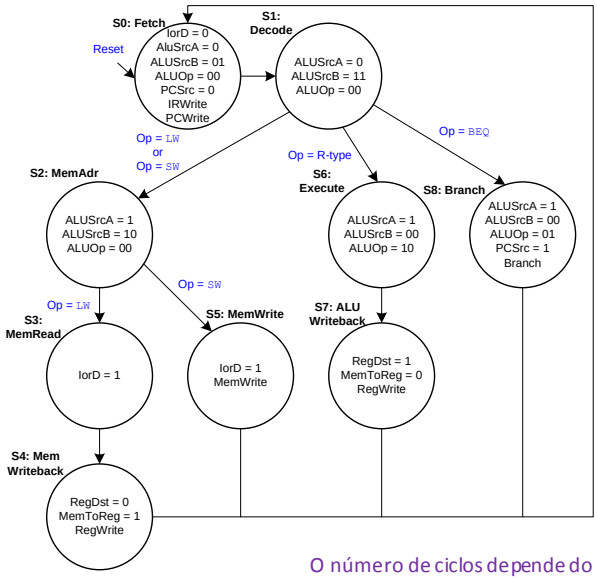
S8: **ALU**: Compara os valores de Rs (A) de Rt (B):
ALUOp = 01 -> ALUControl = 110 (*sub*). Se A e B são iguais Zero=1.
FSM: Caso a instrução (*opcode*) seja beq gera o sinal Branch=1, fazendo PCEn=1.
PCSrc=1 -> PC' = BTA.

FSM (23) - beq (4): Timing - 3 ciclos



A ALU é usada nos 3 ciclos:
S0: Calcula PC+4
S1: Calcula o BTA
S8: Compara A com B

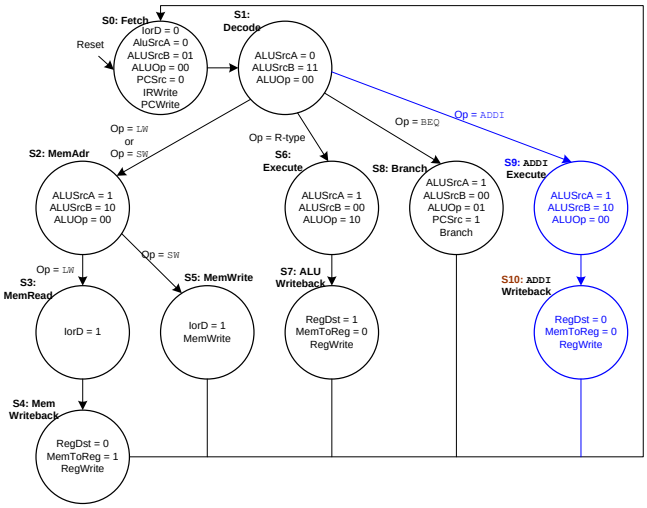
FSM (24) - Completo (s/ Extensões)



O número de ciclos depende do tipo de instrução!

Extensão (1) - addi: FSM (1)

P: Modifique o CPU multicycle para suportar addi?



R:
O *datapath* já é capaz de adicionar o conteúdo dum registo com o valor imediato!

Só precisamos de adicionar novos estados à FSM do controlador para addi.

Os estados são semelhantes aos usados pelas instruções do tipo-R.

S9: Ao registo A é somado SignIm sendo o resultado guardado em ALUout.

S10: ALUout é escrito no RF.

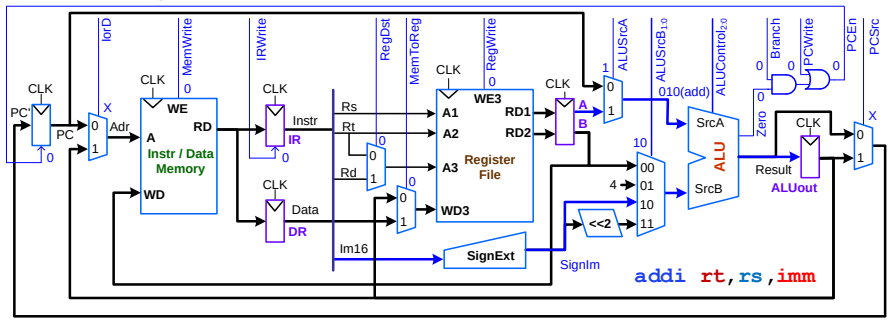
`addi rt,rs,imm`

Extensão (2) - addi: FSM (2) - S9: ALU Exec

S9: Calcula o resultado da operação na ALU

Seleção e ALUOp: ALUSrcA = 1 -> SrcA = A
ALUSrcB = 10 -> SrcB = SignIm
ALUOp = 00 -> ALUControl = 010 (*add*).

Result = A *add* SignIm

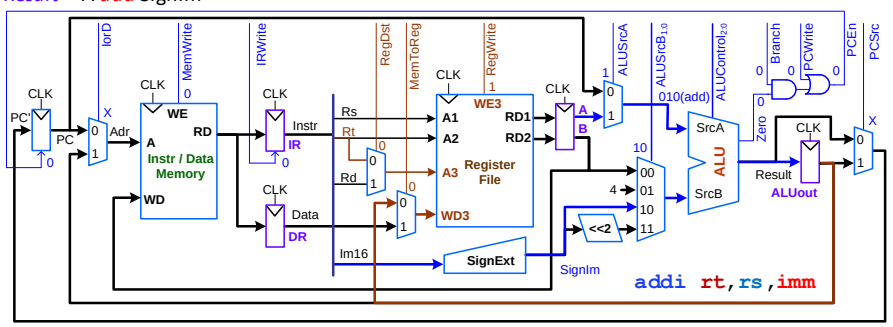


Extensão (3) - addi: FSM (3) - S9 + S10: ALUWriteB

S9: Calcula o resultado da operação na ALU

Seleção e ALUOp: ALUSrcA = 1 -> SrcA = A
ALUSrcB = 10 -> SrcB = SignIm
ALUOp = 00 -> ALUControl = 010 (*add*).

Result = A *add* SignIm



S10: Escrita do valor em ALUout no Banco de Registos

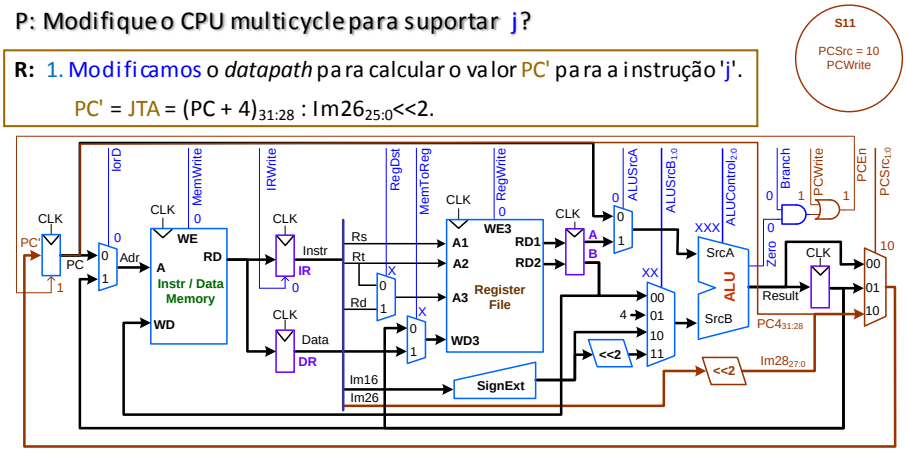
RegDst = 0 -> A3 = Rt e MemToReg = 0 -> WD3 = ALUout.
RegWrite é activado para escrever, completando a execução da instrução.

Extensão (4) - j: Datapath

P: Modifique o CPU multicycle para suportar j?

R: 1. Modificamos o *datapath* para calcular o valor PC' para a instrução 'j'.

PC' = JTA = $(PC + 4)_{31:28}$: $Im26_{25:0}$<<2.

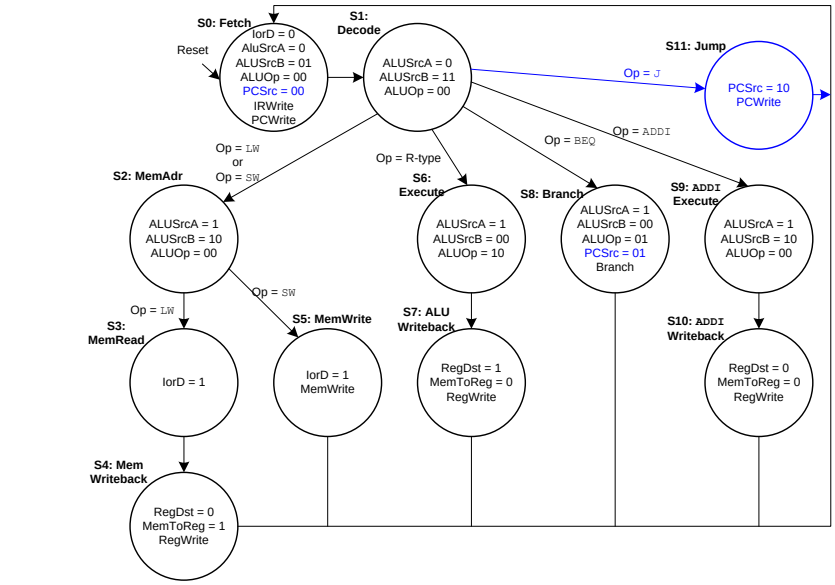


R: 2. O multiplexer PCSrc é estendido para aceitar uma terceira entrada, fazendo PCSrc=10.

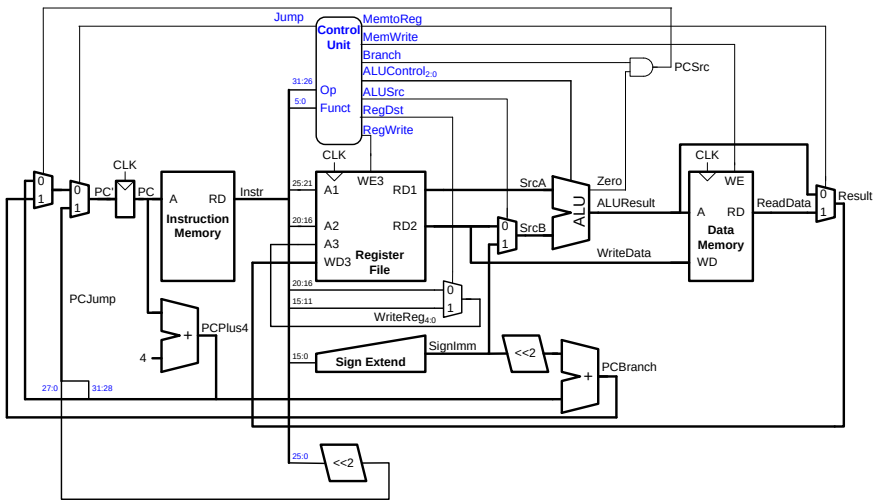
3. PCWRite também tem de ser gerado para esta instrução -> PCEn=1.

4. Adicionamos um estado (S11) à FSM para gerar PCSrc=10 e PCWrite.

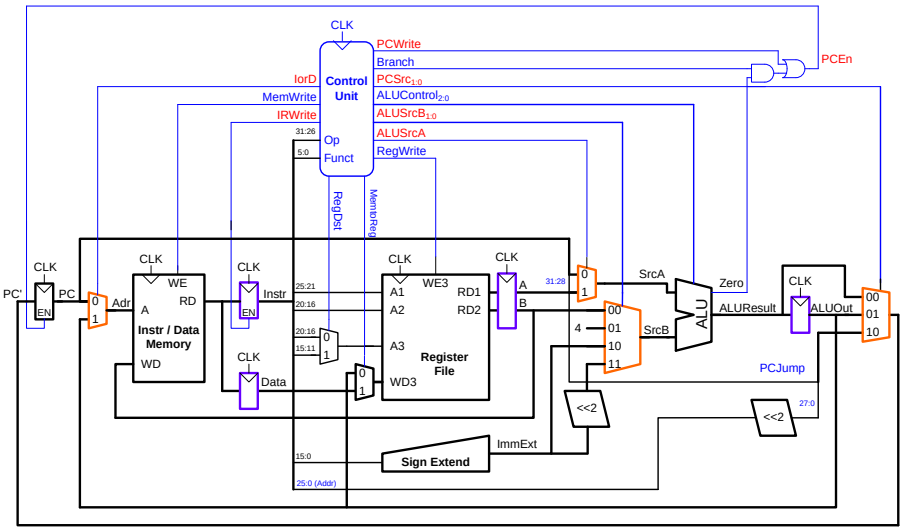
Extensão (5) - j: FSM



Revisão (1) - MIPS Single-Cycle



Revisão (2) - MIPS Multicycle

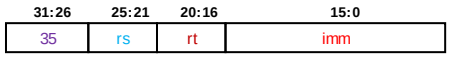


*O *instruction set* suporta: lw/sw, tipo-R, beq + addi e j.

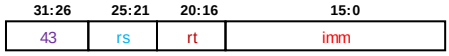
Revisão (3) - Sumário Formato das Intruções (1)

tipo-I

Memory access:

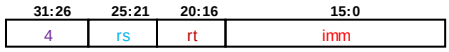


`lw rt, imm(rs)`



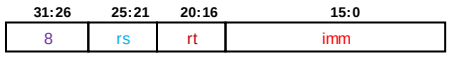
`sw rt, imm(rs)`

Branch:



`beq rs,rt,imm`

Compact Arithmetic/Logic ops:

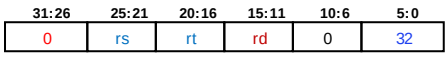


`addi rt,rs,imm`

Revisão (4) - Sumário Formato das Intruções (2)

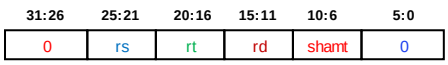
tipo-R

ALU ops: rs e rt



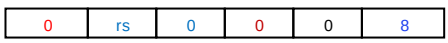
`add rd, rs, rt`

ALU ops: rt e shamt (rs ignored)



`sll rd,rt,shamt`

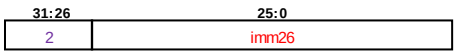
ALU ops: none ; jump to rs contents



`jr rs`

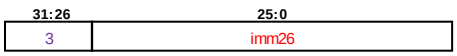
tipo-J

jump:



`j imm26`

jump and link (set $ra):



`jal imm26`