Parte 3
Seção de Dados e Unidade de Controle

Bibliografia

[1] Miles J. Murdocca e Vincent P. Heuring, “Introdução à Arquitetura de Computadores”

Princípios Básicos da Microarquitetura

ISA: conjunto de instruções que efetua operações em registradores e memória

CPU: parte da máquina responsável por implementar estas operações

Microarquitetura:

nível de controle (microprogramado ou fixo em hardware) da CPU

(um ISA pode ser implementado em diferentes microarquiteturas)

O Ciclo de Busca-Execução

- **Passos da Unidade de Controle durante a execução de um programa:**

  (1) Buscar na memória a próxima instrução a ser executada.

  (2) Decodificar o opcode.

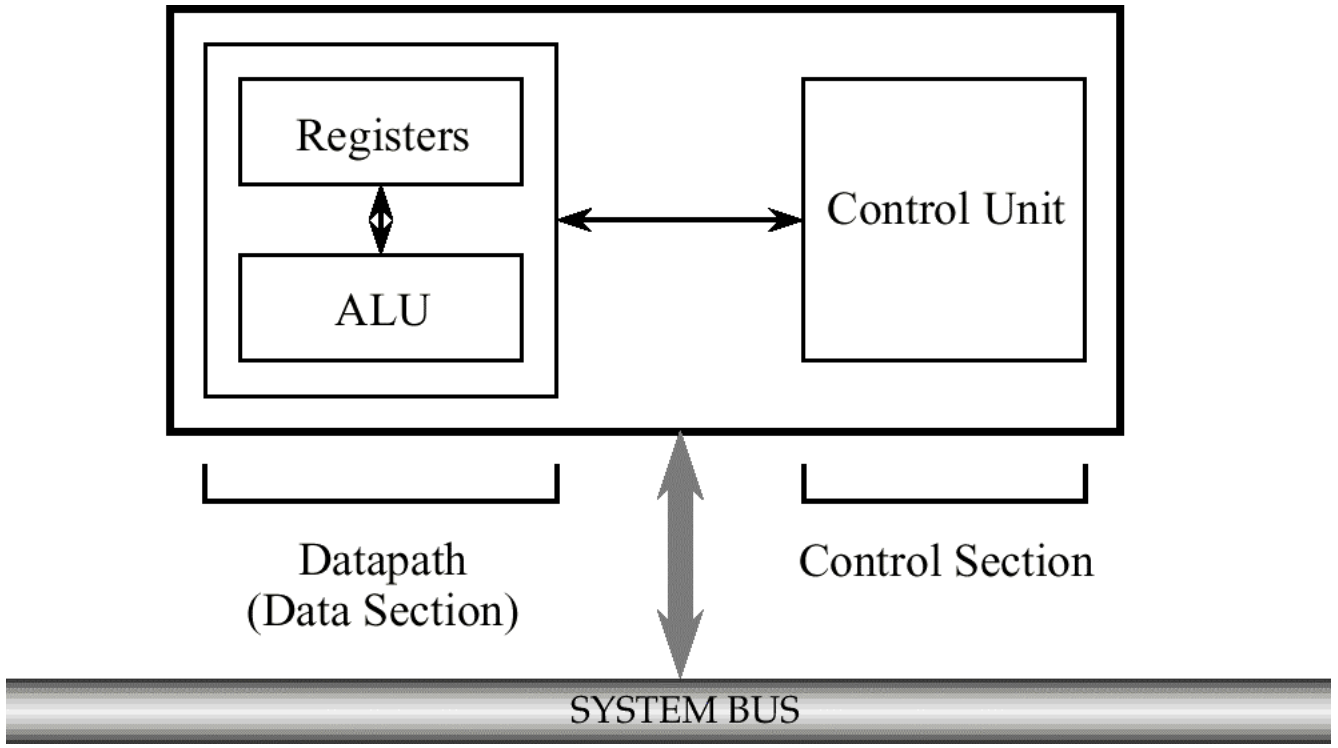
  (3) Ler operandos da memória principal, se houver.

  (4) Executar a instrução e armazenar o resultado.

  (5) Ir para o Passo 1.

Visão de Alto Nível da Microarquitetuta

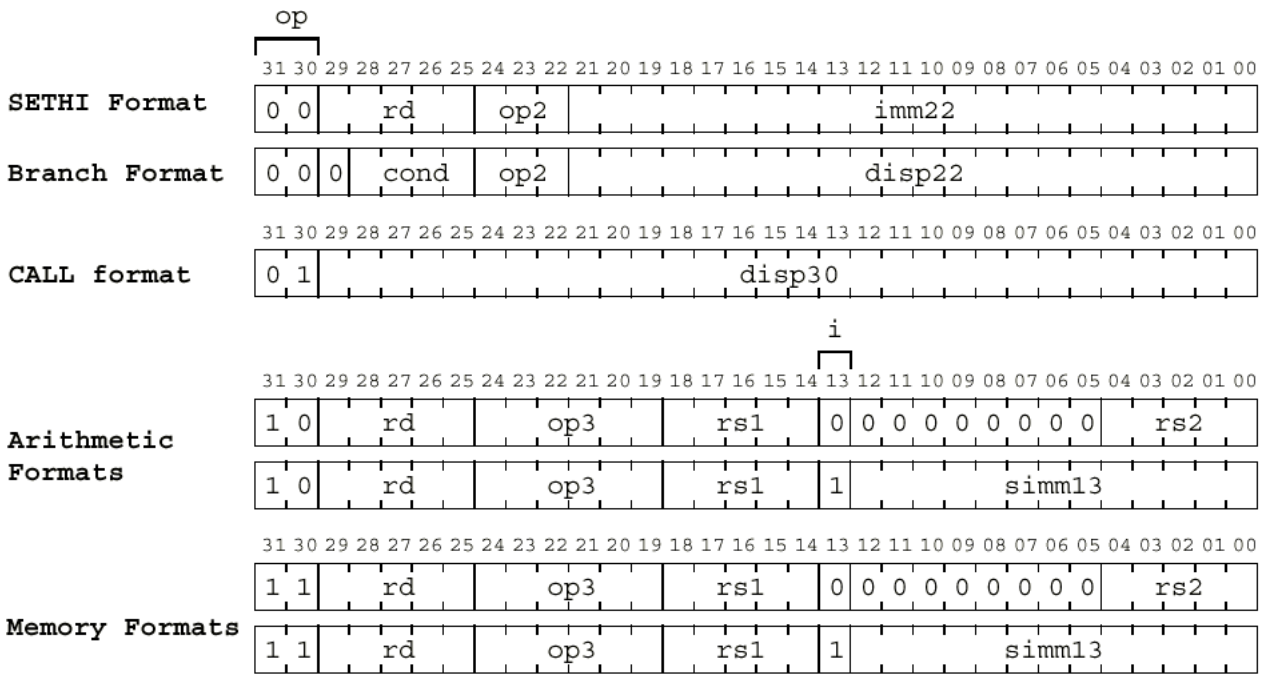
- **A microarquitetura consiste de uma *unidade de controle* e de *registradores* visíveis pelo programador, unidades funcionais como a *ALU*, e quaisquer registradores adicionais necessários à unidade de controle.**



Conjunto de Instruções ARC

	Mnemonic	Meaning
Memory	ld	Load a register from memory
	st	Store a register into memory
Logic	sethi	Load the 22 most significant bits of a register
	andcc	Bitwise logical AND
	orcc	Bitwise logical OR
	orncc	Bitwise logical NOR
	srl	Shift right (logical)
Arithmetic	addcc	Add
Control	call	Call subroutine
	jmpl	Jump and link (return from subroutine call)
	be	Branch if equal
	bneg	Branch if negative
	bcs	Branch on carry
	bvs	Branch on overflow
	ba	Branch always

Formato de Instruções ARC



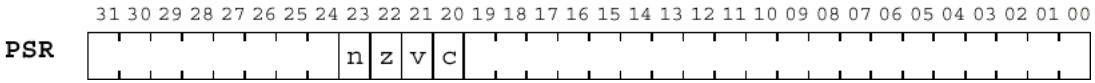
op	Format
00	SETHI/Branch
01	CALL
10	Arithmetic
11	Memory

op2	Inst.
010	branch
100	sethi

op3 (op=10)	
010000	addcc
010001	andcc
010010	orcc
010110	orncc
100110	srl
111000	jmpl

op3 (op=11)	
000000	ld
000100	st

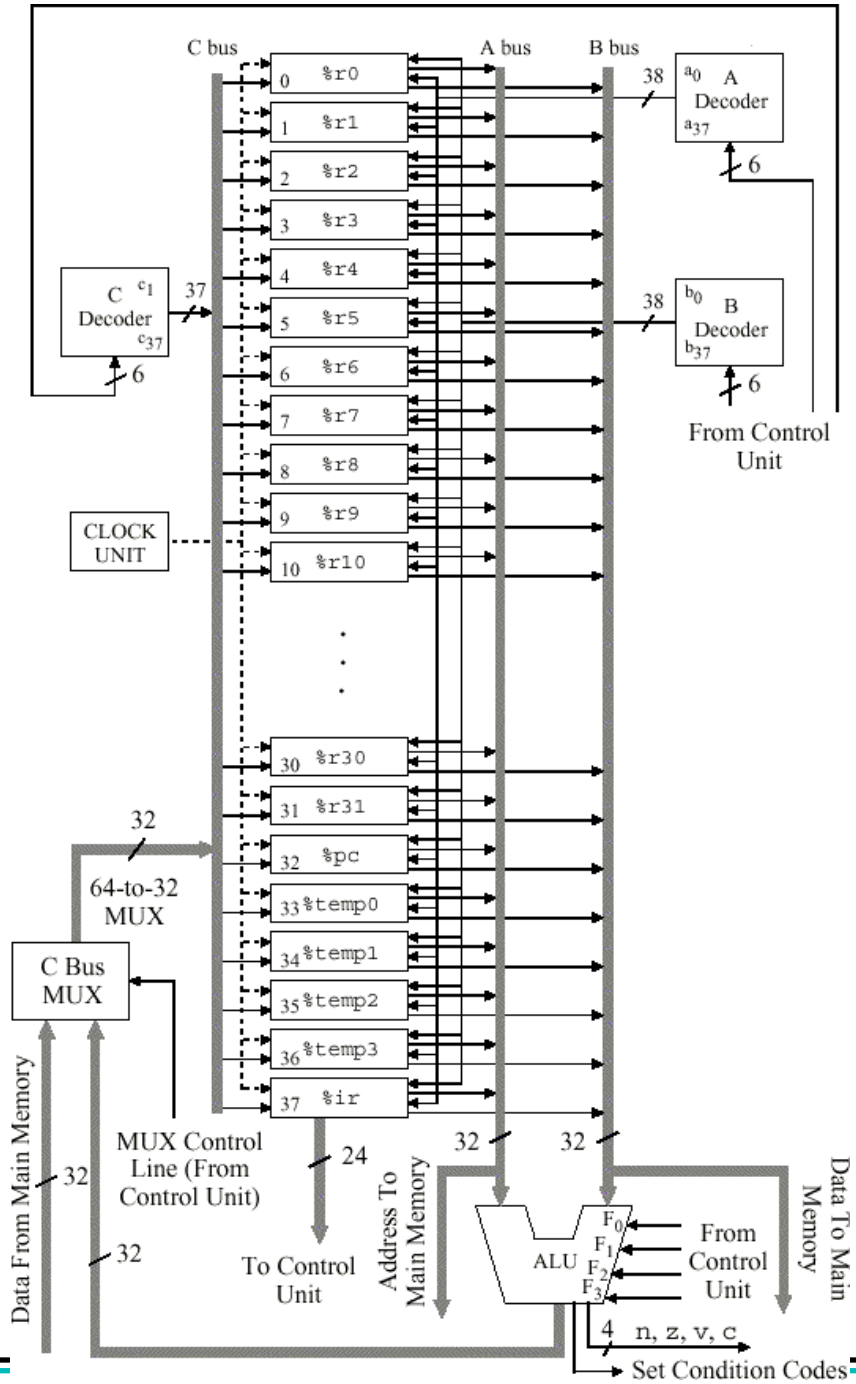
cond	branch
0001	be
0101	bcs
0110	bneg
0111	bvs
1000	ba



Seção de Dados

- Registradores
- ALU (Unidade Lógica Aritmética)
- Conexões entre registradores e ALU, entradas e saídas de controle

Seção de Dados (Datapath) ARC



Registradores ARC

- 32 registradores de dados *visíveis* pelo usuário
  - %r0 a %r31
    - %r0 contém SEMPRE zero e não pode ser modificado
- Contador de programa - %pc
  - Aponta a instrução a ser lida da memória principal
    - Usuário tem acesso através de call e jmpl
- Registrador de instrução - %ir
- 4 registradores temporários *não visíveis* no nível ISA
  - %temp0 a %temp3
    - usados na interpretação das instruções ARC

Operações da ALU ARC

F_3 F_2 F_1 F_0	Operation	Changes Condition Codes
0 0 0 0	ANDCC (A, B)	yes
0 0 0 1	ORCC (A, B)	yes
0 0 1 0	NORCC (A, B)	yes
0 0 1 1	ADDCC (A, B)	yes
0 1 0 0	SRL (A, B)	no
0 1 0 1	AND (A, B)	no
0 1 1 0	OR (A, B)	no
0 1 1 1	NOR (A, B)	no
1 0 0 0	ADD (A, B)	no
1 0 0 1	LSHIFT2 (A)	no
1 0 1 0	LSHIFT10 (A)	no
1 0 1 1	SIMM13 (A)	no
1 1 0 0	SEXT13 (A)	no
1 1 0 1	INC (A)	no
1 1 1 0	INCPC (A)	no
1 1 1 1	RSHIFT5 (A)	no

Operações da ALU

- `ANDCC` e `AND` – "e" lógico bit a bit dos valores nos barramentos A e B
- `ORCC` e `OR` – "ou" bit a bit dos valores de A e B
- `NORCC` e `NOR` – "nou" bit a bit dos valores de A e B
- `ADDCC` e `ADD` – adição em complemento a dois de A e B

- `SRL` – desloca o conteúdo de A para a direita, pela quantidade de bits indicada em B (de 0 a 31 bits)
  - zeros copiados nas posições à esquerda do resultado
- `LSHIFT2` e `LSHIFT10` – deslocam o conteúdo de A para a esquerda por 2 ou 10 bits
  - zeros copiados nas posições mais à direita

Operações da ALU (cont.)

`SIMM13` – recupera 13 bits menos significativos de A e zera outros 19 bits

`SEXT13` – faz uma extensão de sinal dos 13 bits menos significativos de A

`INC` – incrementa o valor de A de 1

`INCPC` – incrementa o valor de A de 4

(serve para incrementar o PC por uma palavra)

`RSHIFT5` – desloca A para a direita de 5 bits, copiando o bit de sinal nos 5 bits à esquerda

Operações da ALU (cont.)

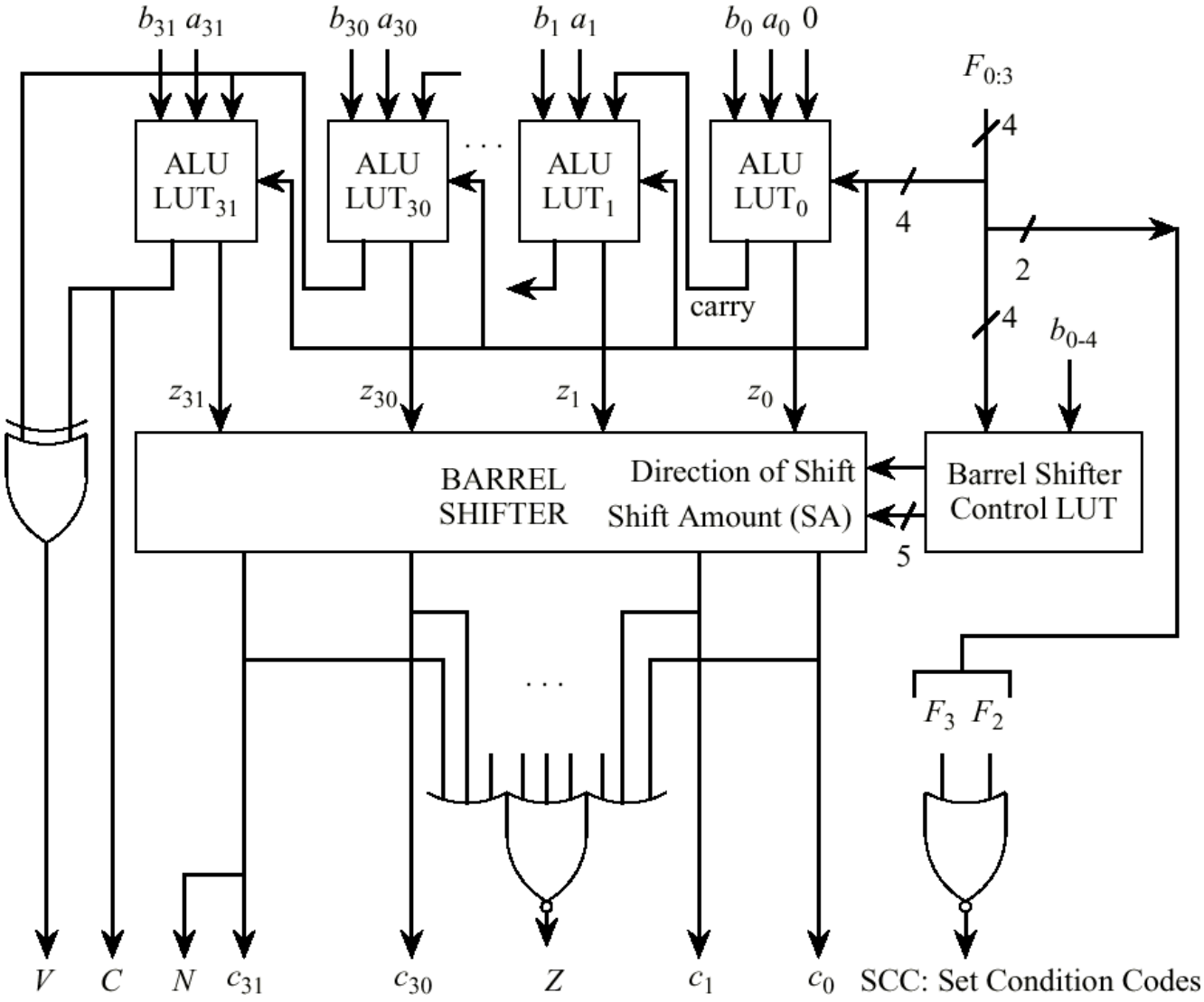
- Cada operação aritmética e lógica pode ser implementada com as operações básicas da ALU

- Códigos de condição gerados pela ALU
  - c – excedente ou vai-um ou *carry*
  - n – negativo
  - z – zero
  - v – *overflow*
  - Sinal SCC (*Set Condition Codes*) faz com que %PSR atualize seu valor

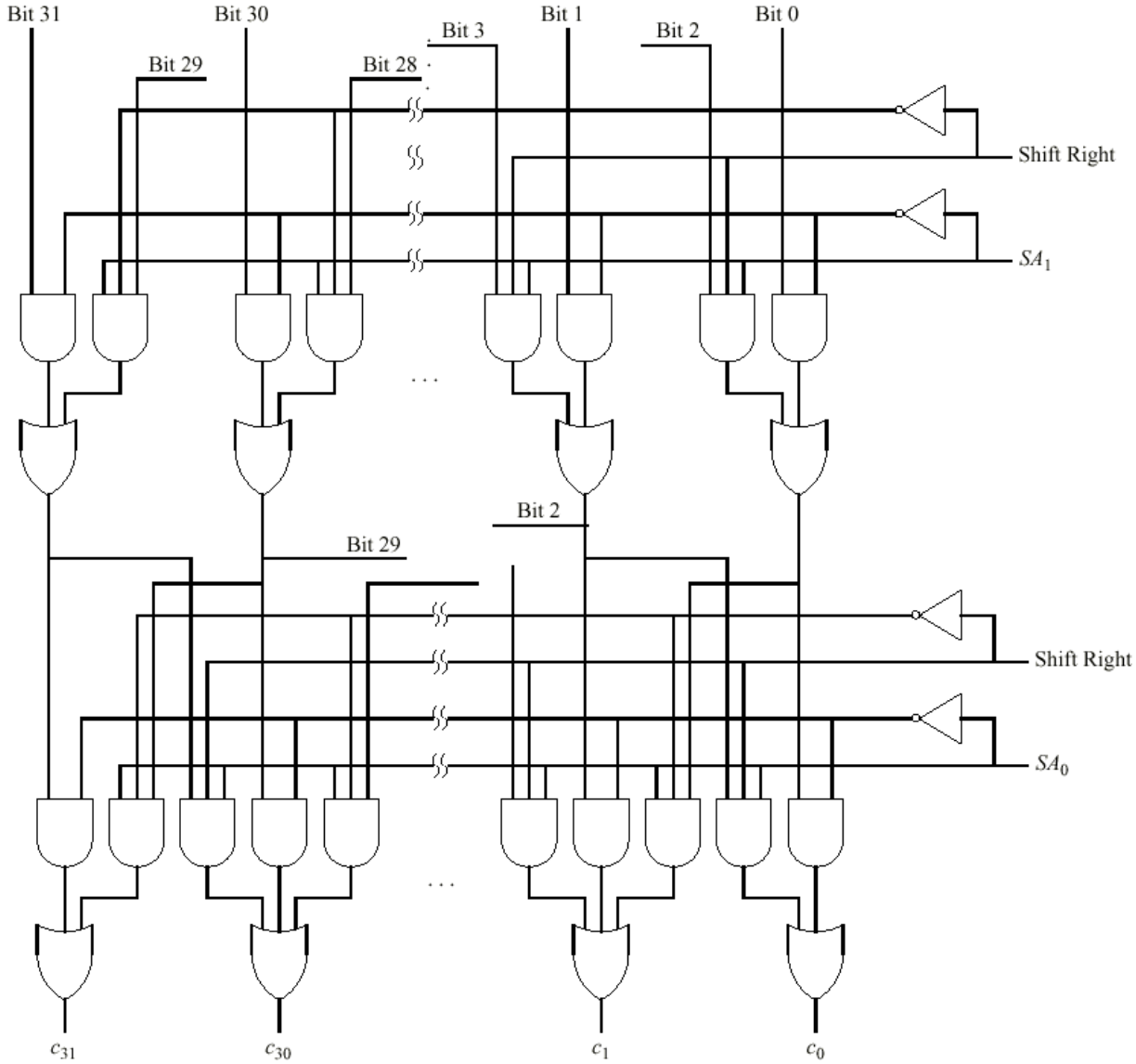
Composição da ALU

- Entradas
  - A e B com 32 bits
  - Entrada de controle F com 4 bits
- Saídas
  - Saída C com 32 bits
  - Código de Condição com 4 bits (n,c,v,z)
  - Sinal SCC
- 32 Tabelas de previsão (LUT – *LookUp Table*)
- Um *barrel shifter* (circuito de deslocamento)

Diagrama de Blocos da ALU



Nível de Portas do *Barrel Shifter*



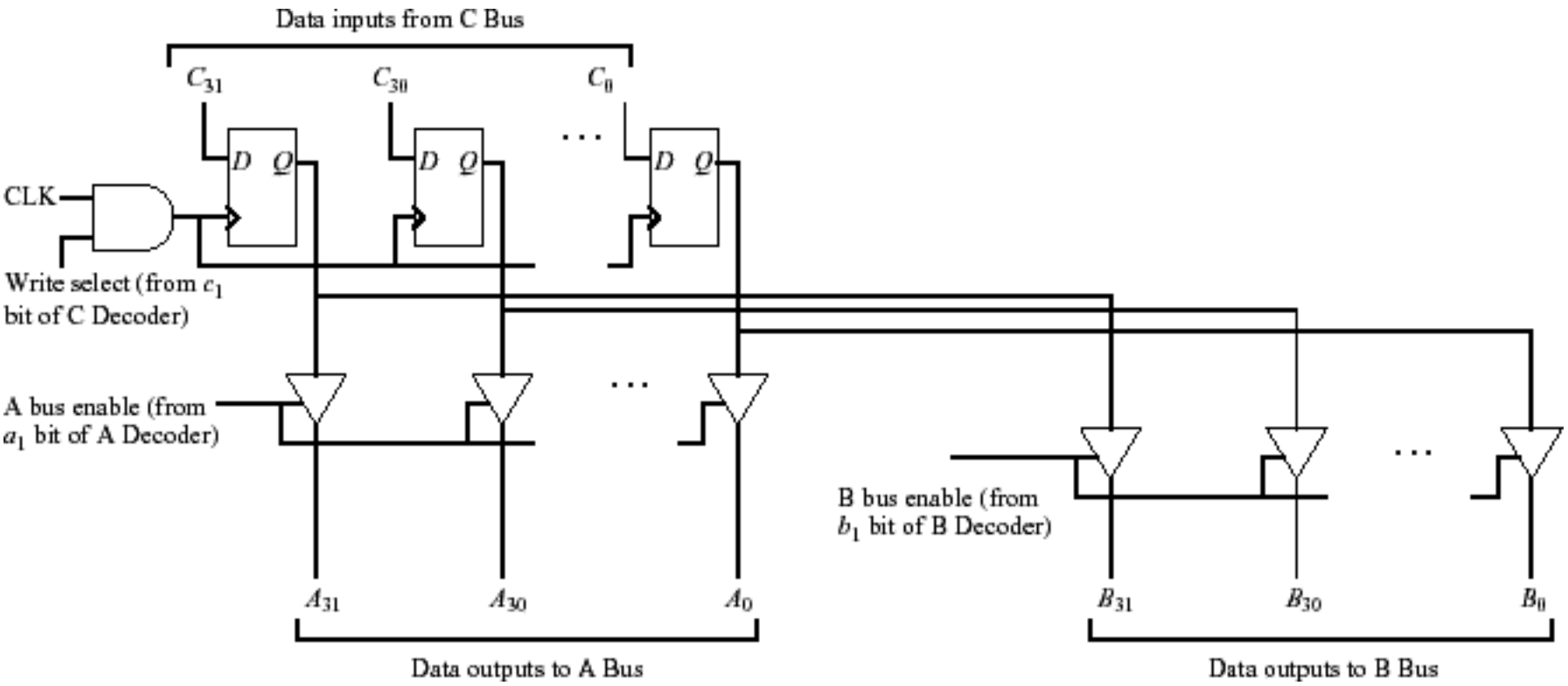
Composição das LUTs

- Entradas
  - $F_0..F_3$
  - b_i, a_i, c_{in} (entrada de excedente ou “vem-um”)
- Saídas
  - z_i, c_{out} (saída de excedente ou vai-um)

Tabela-verdade para (a maior parte das) LUTs da ALU

	F_3	F_2	F_1	F_0	*Carry In*	a_i	b_i	z_i	*Carry Out*
ANDCC	0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	1	0	0
	0	0	0	0	0	1	0	0	0
	0	0	0	0	0	1	1	1	0
	0	0	0	0	1	0	0	0	0
	0	0	0	0	1	0	1	0	0
	0	0	0	0	1	1	0	0	0
	0	0	0	0	1	1	1	1	0
ORCC	0	0	0	1	0	0	0	0	0
	0	0	0	1	0	0	1	1	0
	0	0	0	1	0	1	0	1	0
	0	0	0	1	0	1	1	1	0
	0	0	0	1	1	0	0	0	0
	0	0	0	1	1	0	1	1	0
					.				.
					.				.
					.				.

Projeto do Registrador %r1



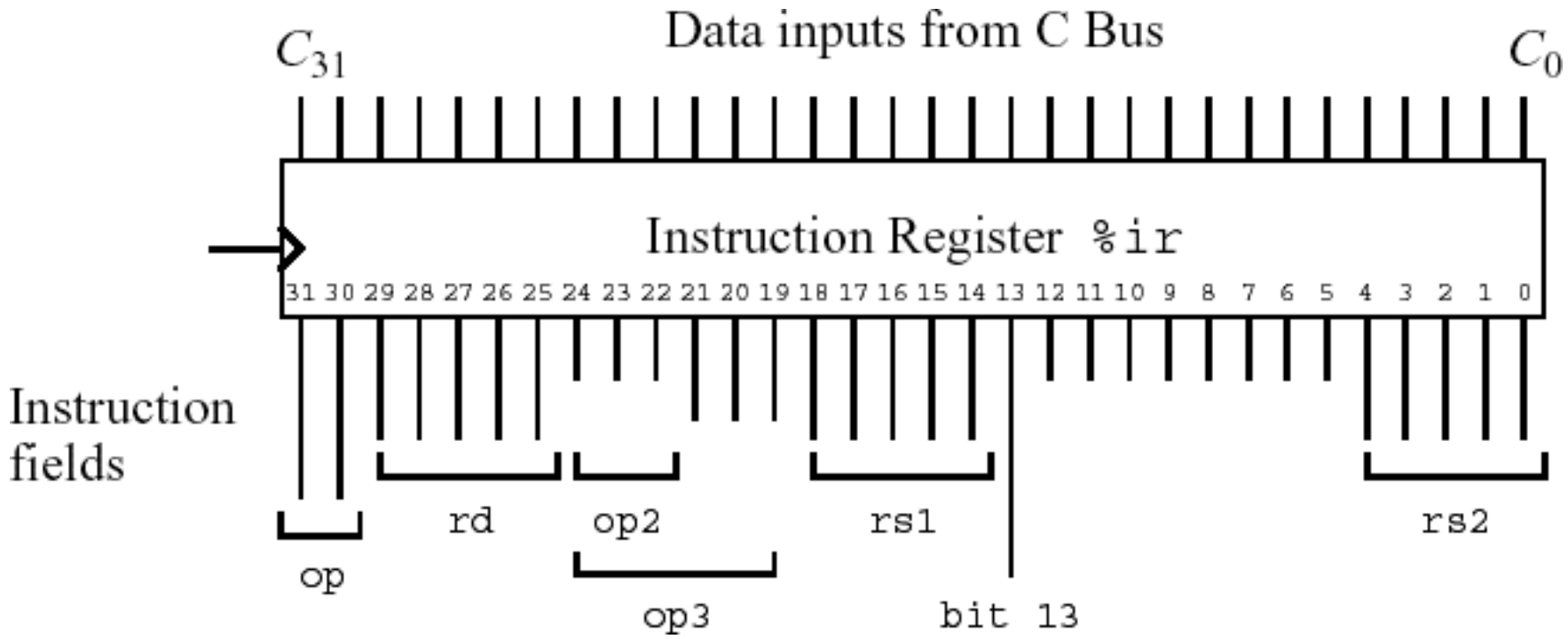
Projeto do Registrador %r1

- Entrada
  - Write Select – AND da linha de seleção c_1 do decodificador C com o sinal de clock
  - C_{31} .. C_0 – entrada de dados do barramento C
- Saída
  - Escritas nos barramentos A e B
  - Habilitação dos barramentos
    - Linha a_1 (b_1) do decodificador A (B)

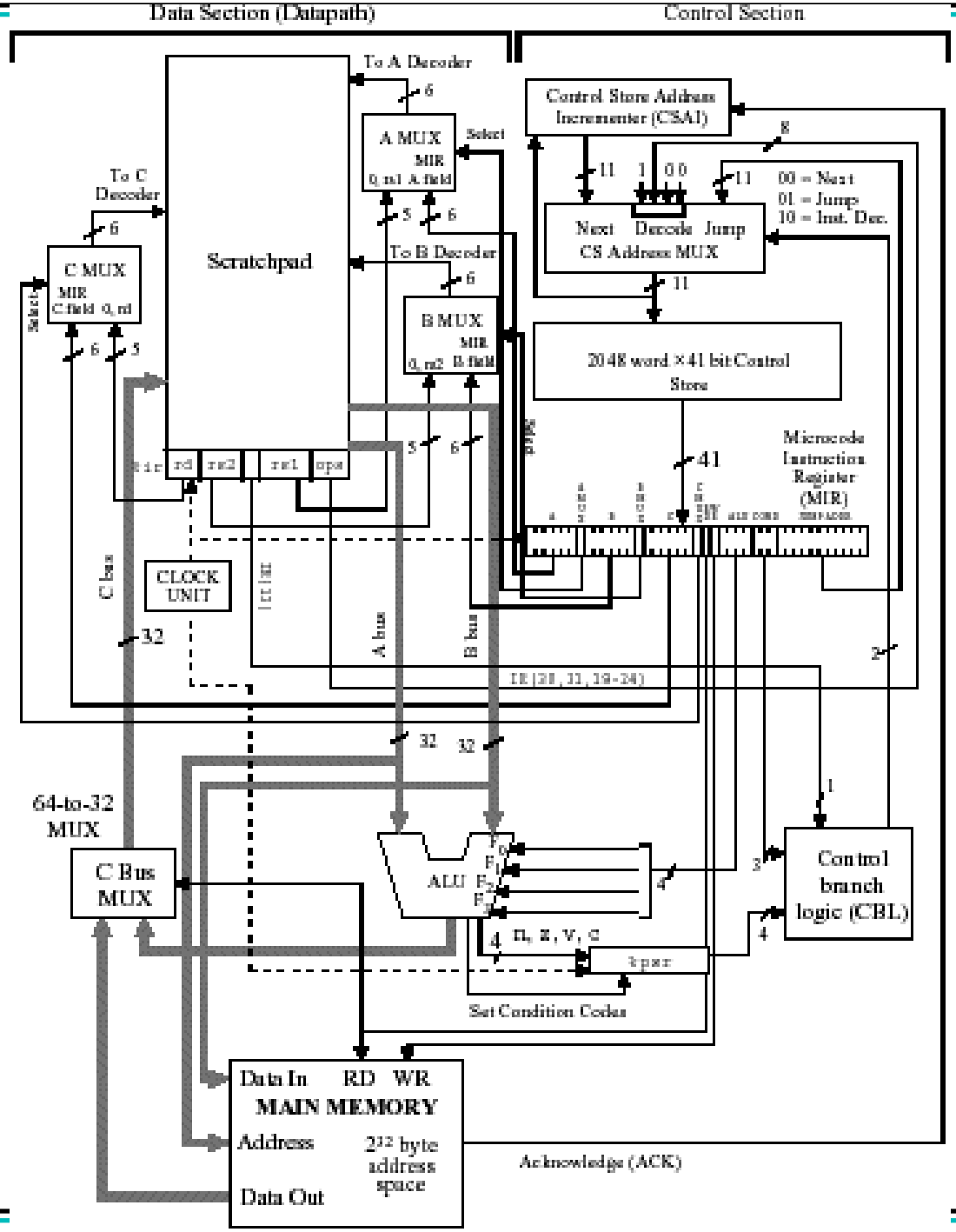
Projeto dos Outros Registradores

- Todos semelhantes a `%r1`
- Exceções
  - `%r0`
    - Não tem entradas vindas do barramento C nem do decodificador C
  - `%ir`
    - Linhas de saída adicionais para a unidade de controle, correspondentes aos campos rd, rs1, rs2, op, op2, op3 e bit 13 de uma instrução
  - `%pc`
    - dois últimos bits podem ser fixos em zero

Saídas para Unidade de Controle vindas do Registrador %ir



Microarquitetura (microprogramada) do ARC



Composição da Seção de Controle

- **Armazenamento de Controle**
  - **Contém o microprograma (um *firmware*)**
  - **Memória ROM (2048 palavras de 41 bits)**
  - **Cada palavra de 41 bits = micro-instrução ou micro-palavra**
  - **Os 41 bits contém valores *para todas as linhas que devem ser controladas*, para implementar as instruções do nível de usuário**

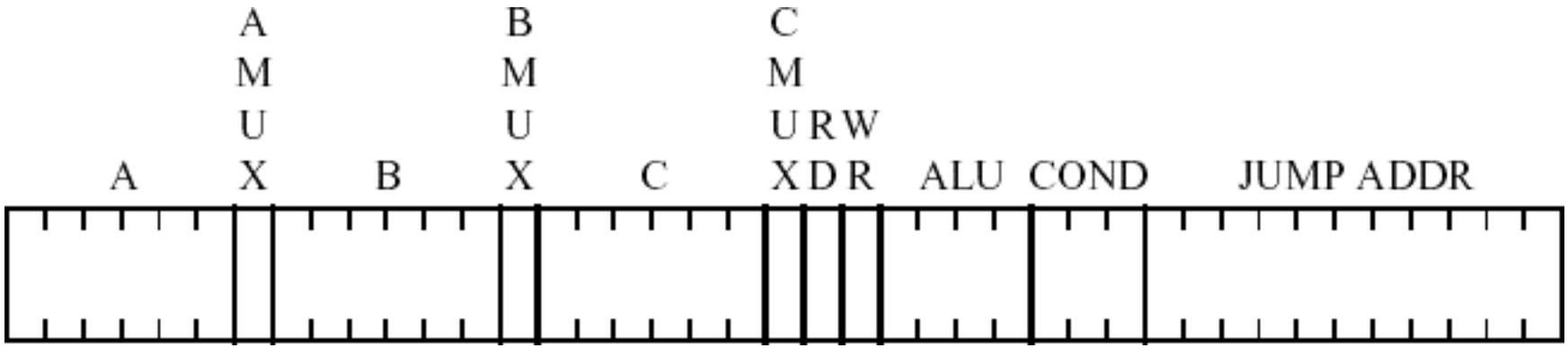
Funcionamento da Unidade de Controle

- Unidade de controle busca e executa micro-instruções
- Execução controlada pelo
  - Registrador de Instrução de Micro-programa (MIR – *Microprogram Instruction Register*)
  - Registrador de Status do Processador (`%psr`)
  - Unidade Lógica de Controle de Desvio (CBL – *Control Branch Logic*)
  - Multiplexador de Endereços do Armazenamento de Controle

Funcionamento da Unidade de Controle

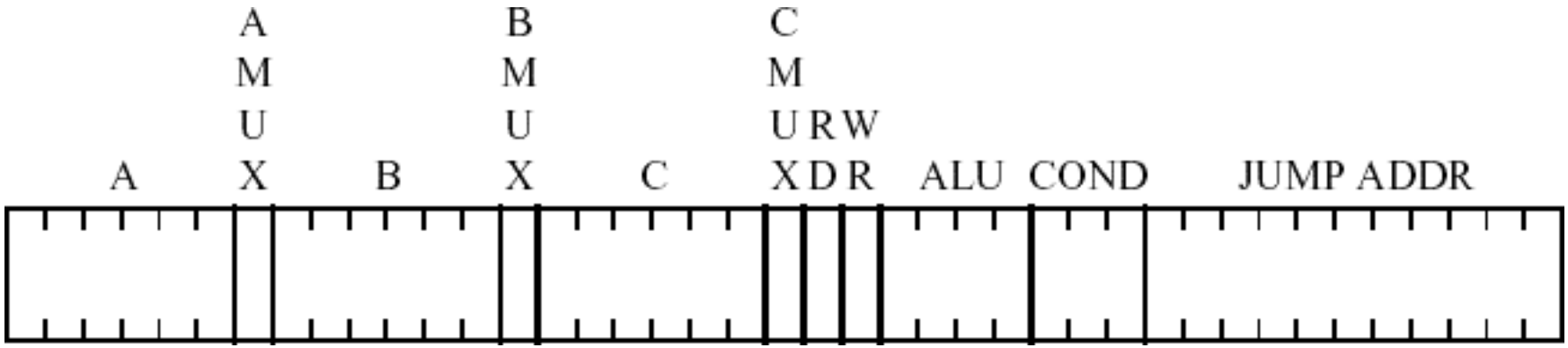
- Início
  - Micro-palavra da posição 0 buscada do Armazenamento de Controle para o registrador MIR e execução
  - Daí em diante, outra micro-palavra é selecionada

Formato da Micro-palavra



- **A**
  - **determina qual registrador da seção de dados será colocado no barramento A**
- **AMUX**
  - **=0 : decodificador A recebe entrada do campo A do MIR**
  - **=1 : decodificador A recebe como entrada 0,rs1 (campo do `%ir`)**
- **B : idem para barramento B**
- **BMUX : idem para campo rs2 do `%ir`**

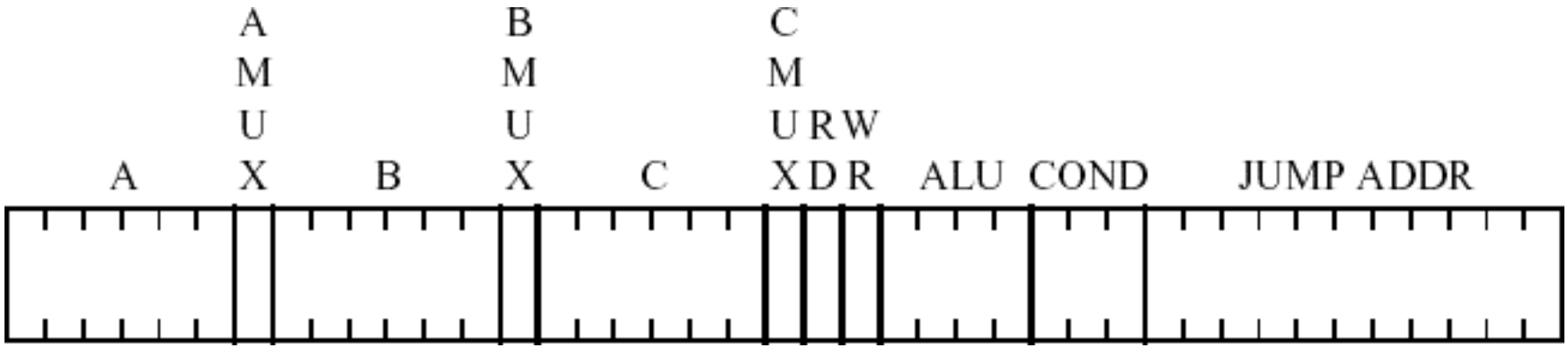
Formato da Micro-palavra



- **C**
  - **determina qual registrador da seção de dados será colocado no barramento C**
- **CMUX**
  - **=0 : decodificador A recebe entrada do campo C do MIR**
  - **=1 : decodificador A recebe como entrada 0,rd (campo do `%ir`)**

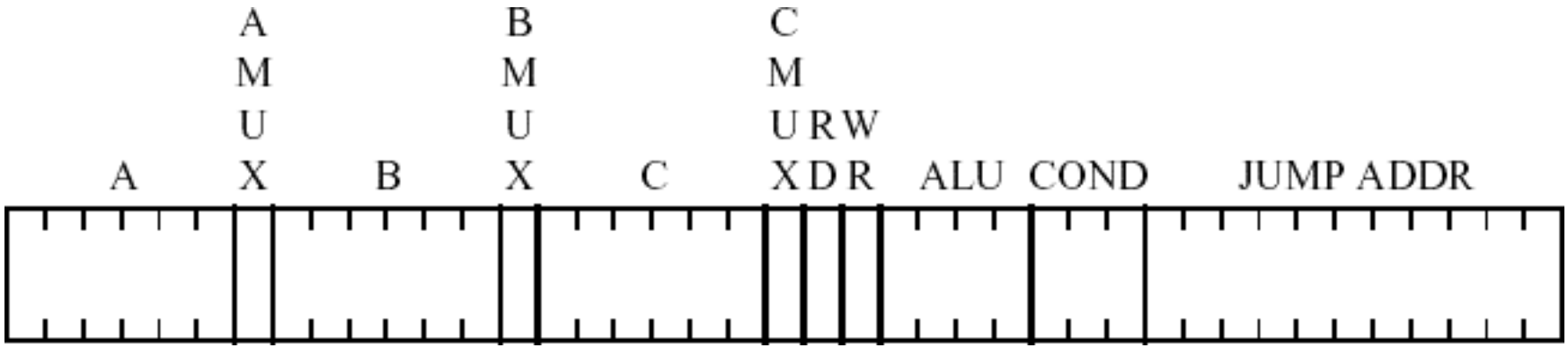
(obs.: como `%r0` é sempre zero, o padrão 000000 pode ser usado no campo C quando nenhum registrador é modificado)

Formato da Micro-palavra



- **RD e WR**
  - **Determinam se a memória vai ser lida ou escrita**
  - **Endereço da memória retirado do barramento A**
  - **Entrada de dados retirada do barramento B**
  - **Saída de dados colocada no barramento C**
  - **RD controla o MUX 64-to-32 do barramento C**
    - **RD =1 barramento C carregado da memória**
    - **RD=0 barramento C carregado da ALU**

Formato da Micro-palavra



- **ALU**
  - **determina a operação da ALU a ser efetuada**
  - **todos os 16 padrões são válidos...**
- **COND – desvio condicional**
  - **próxima palavra:**
    - **próxima posição do armazenamento de controle**
    - **localização do campo JUMP ADDR (11 bits) do MIR**
    - **bits de opcode da instrução em `%ir`**

Lógica de Controle de Desvio (CBL)

- Entradas
  - COND (campo do MIR)
  - n,z,v,c (campos do PSR)
  - bit 13 do `%ir`
- Saída
  - 2 bits para controlar o MUX de Endereços do Armazenamento de Controle

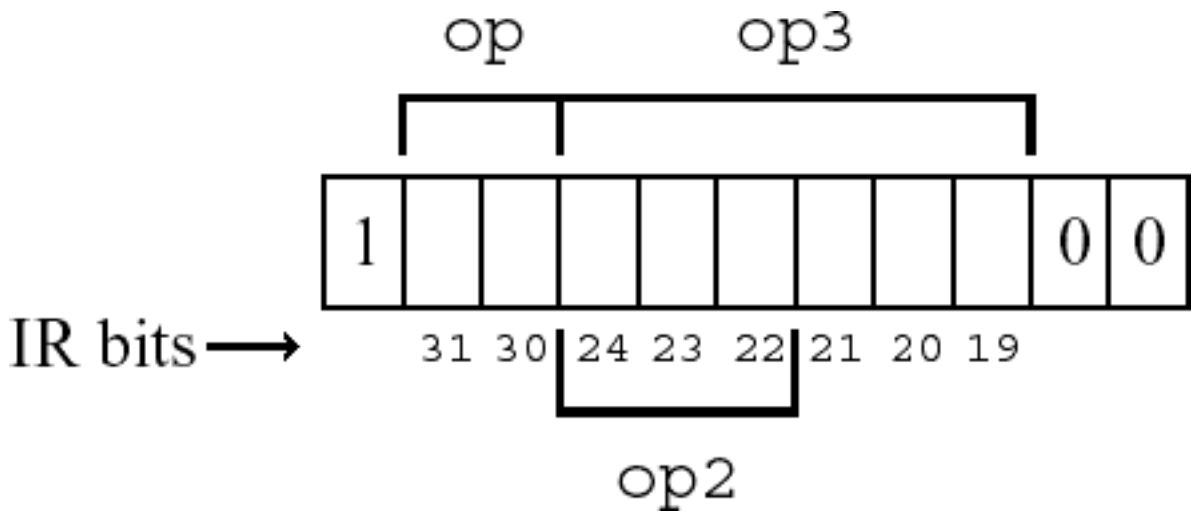
MUX de Endereços do Armazenamento de Controle

- Seletor vindo da CBL (Lógica de Controle de Desvio)
  - 00 = Próximo endereço
    - Entrada = saída do Incrementador de Endereços
  - 01 = Desvie
    - Entrada = JUMP ADDR (vindo do MIR)
  - 10 = Decodifique Instrução
    - Entrada = `1 [campos ops do %ir] 0 0`
  - Saída (11bits)
    - Endereça o Armazenamento de Controle (memória de microprograma)

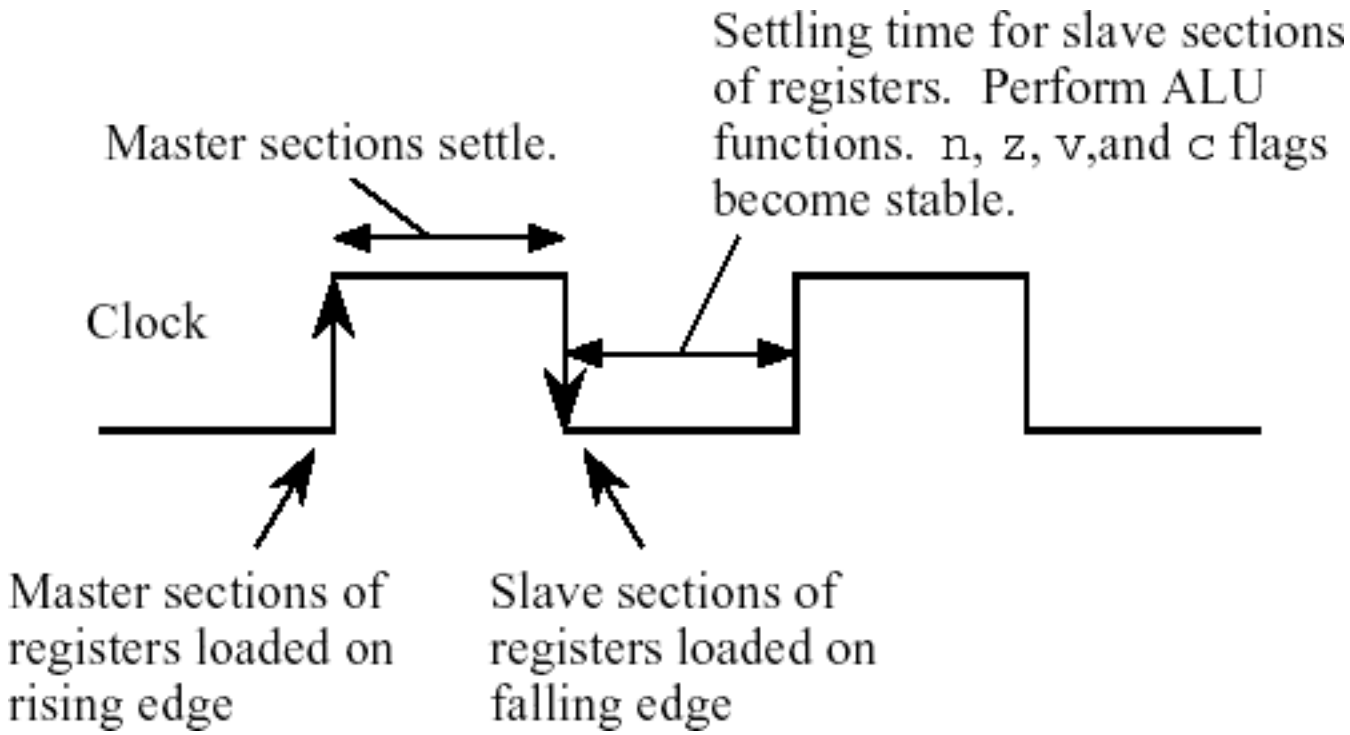
Valores do Campo COND da Micro-palavra

C_2	C_1	C_0	Operation
0	0	0	Use NEXT ADDR
0	0	1	Use JUMP ADDR if n = 1
0	1	0	Use JUMP ADDR if z = 1
0	1	1	Use JUMP ADDR if v = 1
1	0	0	Use JUMP ADDR if c = 1
1	0	1	Use JUMP ADDR if IR[13] = 1
1	1	0	Use JUMP ADDR
1	1	1	DECODE

Formato DECODE para um Endereço de Micro-instrução



Relações de Tempo entre os Registradores



Micro-programa Parcial ARC

```
   0: R[ir] <- AND(R[pc],R[pc]); READ;          /Leia a instr. ARC da mem. principal

   1: DECODE;                                   /Desvio de acordo com opcode

/sethi

1152: R[rd] <- LSHIFT(ir); GOTO 2047;           /Copie campo imm22 para regist. dest.

/call

1280: R[15] <- AND(R[pc],R[pc]);                /Guarde %pc em %r15

1281: R[temp0] <- ADD(R[ir],R[ir]);             /Desloque 2 bits à esquerda para

1282: R[temp0] <- ADD(R[temp0],R[temp0]);       /colocar disp30 em %temp0

1283: R[pc] <- ADD(R[pc], R[temp0]); GOTO 0;    /Desvie para sub-rotina

/addcc

1600: IF R[IR[13]] THEN GOTO 1602;              /A segunda origem é imediata?

1601: R[rd] <- ADDCC(R[rs1],R[rs2]); GOTO 2047;  / Faça ADDCC nos registradores origem

1602: R[temp0] <- SEXT13(R[ir]);                /Leia campo simm13 estendido por sinal

1603: R[rd] <- ADDCC(R[rs1],R[temp0]);          /Faça ADDCC das origens regist./simm13

      GOTO 2047;
```

Micro-programa Parcial ARC

/andcc

```
1604: IF R[IR[13]] THEN GOTO 1606;                  /A segunda origem é imediata?
1605: R[rd] <- ANDCC(R[rs1],R[rs2]); GOTO 2047;     /Faça ANDCC nos registradores origem
1606: R[temp0] <- SIMM13(R[ir]);                    /Leia campo simm13
1607: R[rd] <- ADDCC(R[rs1],R[temp0]); GOTO 2047;   /Faça ANDCC das origens regist./simm13
```

/orcc

```
1608: IF R[IR[13]] THEN GOTO 1610;                  /A segunda origem é imediata?
1609: R[rd] <- ORCC(R[rs1],R[rs2]); GOTO 2047;     /Faça ORCC nos registradores origem
1610: R[temp0] <- SIMM13(R[ir]);                    /Leia campo simm13
1611: R[rd] <- ORCC(R[rs1],R[temp0]); GOTO 2047;   /Faça ORCC das origens regist./simm13
```

/orncc

```
1624: IF R[IR[13]] THEN GOTO 1610;                  /A segunda origem é imediata?
1625: R[rd] <- NORCC(R[rs1],R[rs2]); GOTO 2047;     /Faça NORCC nos registradores origem
1626: R[temp0] <- SIMM13(R[ir]);                    /Leia campo simm13
1627: R[rd] <- NORCC(R[rs1],R[temp0]); GOTO 2047;   /Faça NORCC das origens regist./simm13
```

Micro-programa Parcial ARC

```
/srl
1688: IF R[IR[13]] THEN GOTO 1690;                    /A segunda origem é imediata?
1689: R[rd] <- SRL(R[rs1],R[rs2]); GOTO 2047;         /Faça SRL nos registradores origem
1690: R[temp0] <- SIMM13(R[ir]);                      /Leia campo simm13
1691: R[rd] <- SRL(R[rs1],R[temp0]); GOTO 2047;       /Faça SRL nas origens regist./simm13
/jmpl
1760: IF R[IR[13]] THEN GOTO 1762;                    /A segunda origem é imediata?
1761: R[pc] <- ADD(R[rs1],R[rs2]); GOTO 0;            /Faça ADD nos registradores origem
1762: R[temp0] <- SEXT13(R[ir]);                      /Leia campo simm13 estendido por sinal
1763: R[pc] <- ADD(R[rs1],R[temp0]); GOTO 0;          /Faça ADD das origens regist./simm13
/ld
1792: R[temp0] <- ADD(R[rs1],R[rs2]);                 /End. orig. = %rs1+%rs2 (se i = 0)
      IF IR[13] THEN GOTO 1794;                       /Se i = 1, end. orig. = %rs1+imm13
1793: R[rd] <- AND(R[temp0],R[temp0]);                /Coloque o end. de orig. no bus A
      READ; GOTO 2047;                                /Leia da memória para R[rd]
1794: R[temp0] <- SEXT13(R[ir]);                      /Leia o campo imm13 do end. de orig.
1795: R[temp0] <- ADD(R[rs1],R[temp0]); GOTO 1793;    /End. de orig. = %rs1+imm13
```

Micro-programa Parcial ARC

```
/st

1808: R[temp0] <- ADD(R[rs1],R[rs2]);              /End. destino = %rs1+%rs2 (se i = 0)
      IF IR[13] THEN GOTO 1810;                    /Se i = 1, end. dest. = %rs1+simm13
1809: R[ir] <- RSHIFT5(R[ir]); GOTO 40;            /Mova rd para posição do campo rs2
  40: R[ir] <- RSHIFT5(R[ir]);                     /deslocando à direita 25 bits
  41: R[ir] <- RSHIFT5(R[ir]);
  42: R[ir] <- RSHIFT5(R[ir]);
  43: R[ir] <- RSHIFT5(R[ir]);
  44: R[0] <- AND(R[temp0],R[rs2]);                /Coloque endereço no barramento A
      WRITE; GOTO 2047;                            /e dados no barramento B; escreva
1810: R[temp0] <- SEXT13(R[ir]);                   /Leia campo simm13 do endereço destino
1811: R[temp0] <- ADD(R[rs1],R[temp0]);            /Calcule endereço de destino %rs1+simm13
      GOTO 1809;
```

```
/Instruções de desvio: ba, be, bcs, bvs, bneg
 1088: GOTO 2;                          /Árvore de decodificação para desvios
    2: R[temp0] <- LSHIFT10(R[ir]);     /Estenda por sinal os 22 bits menores de ir
    3: R[temp0] <- RSHIFT5(R[ir]);      /Deslocando à esquerda 10 bits, e depois à
    4: R[temp0] <- RSHIFT5(R[ir]);      /direita 10 bits. RSHIFT estende o sinal.
    5: R[ir] <- RSHIFT5(R[ir]);         /Move campo COND para IR[13], aplicando
    6: R[ir] <- RSHIFT5(R[ir]);         /RSHIFT5 três vezes. (A extensão do sinal é
    7: R[ir] <- RSHIFT5(R[ir]);         /irrelevante.
    8: IF R[IR[13]] THEN GOTO 12;       /É ba?
       R[IR] <- ADD(R[IR],R[IR]);
    9: IF R[IR[13]] THEN GOTO 13;       /Não é be?
       R[IR] <- ADD(R[IR],R[IR]);
   10: IF Z THEN GOTO 12;               /Execute be
       R[IR] <- ADD(R[IR],R[IR]);
   11: GOTO 2047;                       /Desvio para be não tomado
   12: R[pc] <- ADD(R[pc],R[temp0]);    /Desvio tomado
       GOTO 0;
   13: IF R[IR[13]] THEN GOTO 16;       /É bcs?
       R[IR] <- ADD(R[IR],R[IR]);
   14: IF C THEN GOTO 12;               /Execute bcs
   15: GOTO 2047;                       /Desvio para bcs não tomado
   16: IF R[IR[13]] THEN GOTO 19;       /É bvs?
   17: IF N THEN GOTO 12;               /Execute bneg
   18: GOTO 2047;                       /Desvio para bneg não tomado
   19: IF V THEN GOTO 12;               /Execute bvs
   20: GOTO 2047;                       /Desvie para bvs não tomado
 2047: R[pc] <- INCPC(R[PC]); GOTO 0;   /Incremente %pc e comece novamente
```

Execução de um Programa de Usuário

- 1ª tarefa:
  - trazer a instrução apontada por %pc da memória para %ir

- Linha 0 do microprograma
- `0: R[ir] <- AND(R[PC],R[PC]); READ;`
  - PC carregado no barramento A (`AND(R[pc],R[pc]`)
  - Operação de leitura da memória (`READ`)

Obs. Resultado da operação da ALU descartado, pois MUX Bus C está selecionando "dados da memória" com `READ = 1`

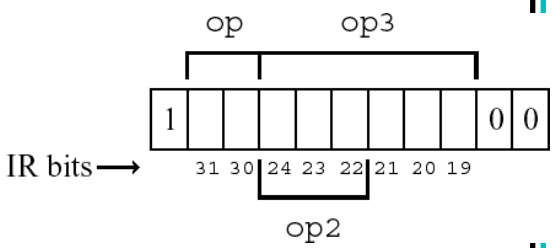
Execução de um Programa de Usuário

Obs.: incremento para próxima instrução só ocorre quando um sinal de confirmação (ACK) for recebido

- **(sinal ligando a Memória ao Incrementador de Endereços de Armazenamento de Controle)**

- **Normalmente, a *próxima* microinstrução é executada, a não ser que haja um `GOTO` ou `DECODE`**

- **Linha 1: `DECODE;` (μ-instrução atual no MIR)**
  - **Decodificar a instrução armazenada em `%ir`**
    - **(decodificação dos opcodes)**
  - **Desvio realizado de acordo com qual das 15 instruções do ARC está sendo interpretada**



Codificação do DECODE

- Para `ADDCC`: op = 10 op3 = 010000

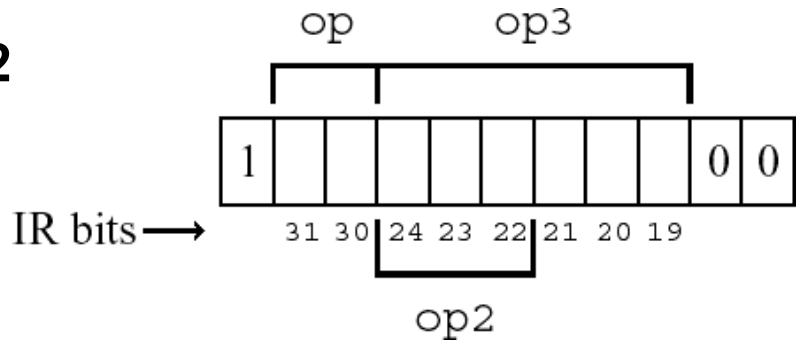
  endereço `DECODE` = 11001000000 = $(1600)_{10}$

- Ou seja, micro-rotina da instrução `ADDCC` começa no endereço 1600 do Armazenamento de Controle

- Alguns endereços de `DECODE` não ocorrem na prática
  - Nestas posições, uma micro-rotina deve ser colocada para tratar da instrução ilegal

Codificação do DECODE (cont.)

- **`SETHI/Branch` possuem op e op2**
- **`CALL` possui apenas op**



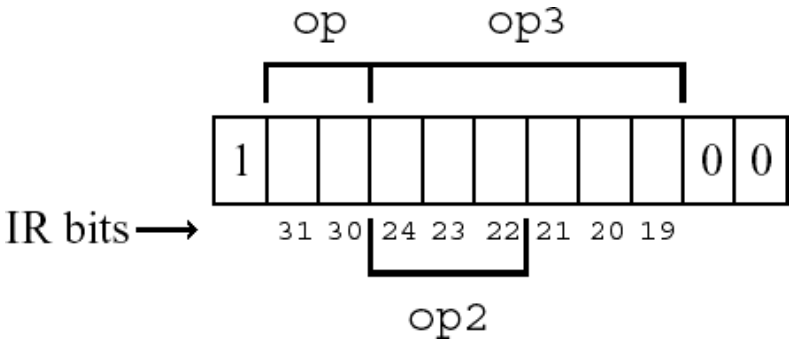
- **`SETHI`: op = 00, op2 = 100 `DECODE = 100100xxx00`**
  - **há duplicação do código `SETHI` nas oito posições de memória correspondentes aos `xxx`**
- **O mesmo é feito para `CALL` e instruções de desvio**

- **Ineficiente do ponto de vista memória, mas codificação rápida e simples**
  - **Opção: modificar o decodificador para o armazenamento de controle**

Exemplo: instrução `ld`

Instrução assembly: `ld %r5+80, %r2`

```
11 00010 000000 00101 1 0000001010000
op  rd    op3    rs1  i    simm13
```



Endereço da micro-instrução = $(1\ 11\ 000000\ 00)_2 = (1792)_{10}$

Exemplo: instrução `ld`

```
11 xxxxx 000000 wwwww 0 00000000 zzzzz
op  rd     op3   rs1              rs2
11 xxxxx 000000 wwwww 1 zzzzzzzzzzzzz
op  rd     op3   rs1        simm13
```

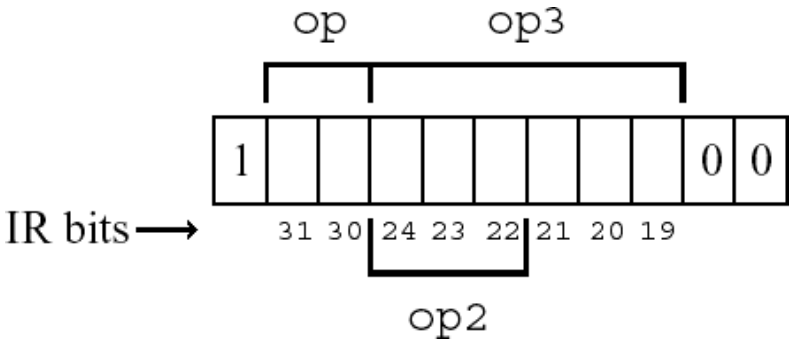
Ex. `ld %r5+80, %r2`

```
   0: R[ir] <- AND(R[pc],R[pc]); READ;   /Leia a instr. ARC da mem. principal
   1: DECODE;                            /Desvio de acordo com opcode
1792: R[temp0] <- ADD(R[rs1],R[rs2]);    /End. orig. = %rs1+%rs2 (se i = 0)
      IF IR[13] THEN GOTO 1794;          /Se i = 1, end. orig. = %rs1+imm13
1794: R[temp0] <- SEXT13(R[ir]);         /Leia o campo imm13 do end. de orig.
1795: R[temp0] <- ADD(R[rs1],R[temp0]);  /End. de orig. = %rs1+imm13
      GOTO 1793;
1793: R[rd] <- AND(R[temp0],R[temp0]);   /Coloque o end. de orig. no bus A
      READ; GOTO 2047;                   /Leia da memória para R[rd]
2047: R[pc] <- INCPC(R[pc]); GOTO 0;     /Incremente o PC, comece novamente
```

Exemplo: instrução `st`

Instrução assembly: `st %r6+32, %r2`

```
11 00010 000100 00110 1 0000000100000
op  rd    op3    rs1  i    simm13
```



Endereço da micro-instrução = $(1\ 11\ 000100\ 00)_2 = (1808)_{10}$

Exemplo: instrução `st`

```
   0: R[ir] <- AND(R[pc],R[pc]); READ;   /Leia a instr. ARC da mem. principal
   1: DECODE;                            /Desvio de acordo com opcode
1808: R[temp0] <- ADD(R[rs1],R[rs2]);    /End. destino = %rs1+%rs2 (se i = 0)
      IF IR[13] THEN GOTO 1810;          /Se i = 1, end. dest. = %rs1+simm13
1809: R[ir] <- RSHIFT5(R[ir]); GOTO 40;  /Mova rd para posição do campo rs2
  40: R[ir] <- RSHIFT5(R[ir]);           /deslocando à direita 25 bits
  41: R[ir] <- RSHIFT5(R[ir]);
  42: R[ir] <- RSHIFT5(R[ir]);
  43: R[ir] <- RSHIFT5(R[ir]);
  44: R[0] <- AND(R[temp0],R[rs2]);  /Coloque endereço no barramento A
      WRITE; GOTO 2047;              / e endereço no barramento B
1810: R[temp0] <- SEXT13(R[ir]);     /Leia campo simm13 do endereço destino
1811: R[temp0] <- ADD(R[rs1],R[temp0]); /Calcule endereço de destino
      GOTO 1809;                       / (%rs1+simm13)
2047: R[pc] <- INCPC(R[pc]); GOTO 0;     /Incremente o PC, comece novamente
```

Decodificação de Instruções de Desvio

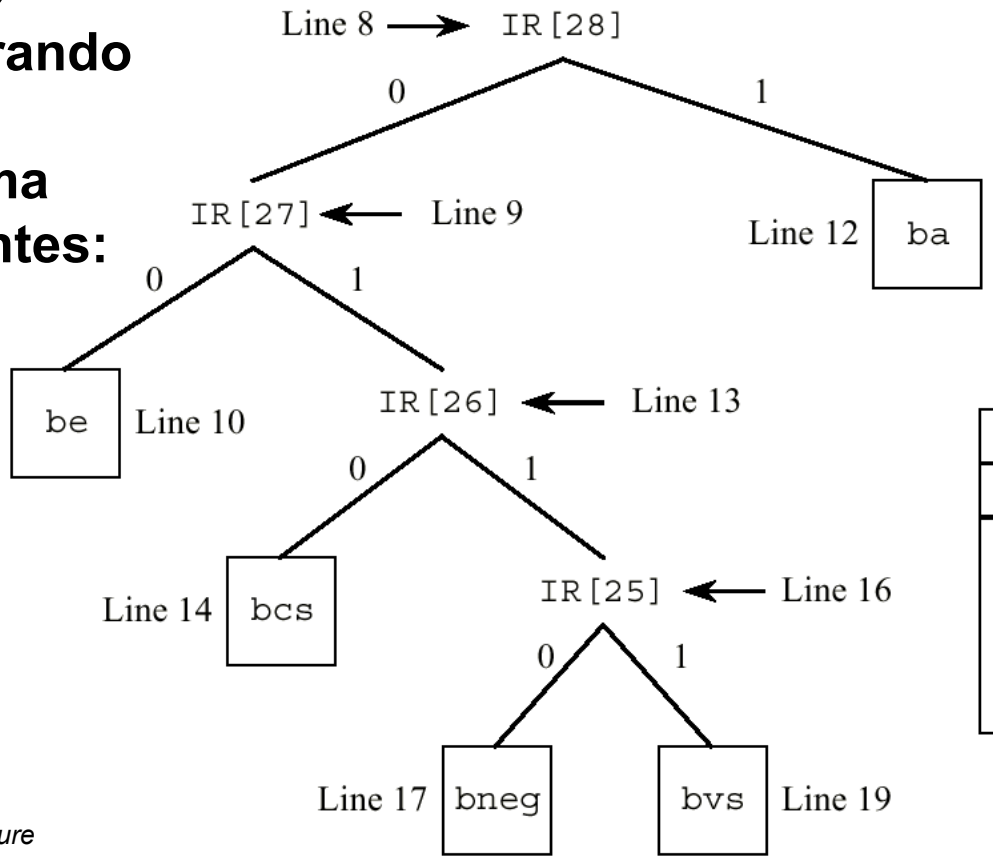
- `DECODE` transfere controle para posição 1088
- Linha 1088 transfere para linha 2, seção de memória especialmente alocada para as instruções de desvio

- Formato de instruções de desvio

```
00 0 xxxx yyy zzzzzzzzzzzzzzzzzzzzzz
op   cond op2    disp22
```

Decodificação de Instruções de Desvio

- **Árvore de decodificação para instruções de desvio, mostrando as linhas de microprograma correspondentes:**



cond				branch
28	27	26	25	
0	0	0	1	be
0	1	0	1	bcs
0	1	1	0	bneg
0	1	1	1	bvs
1	0	0	0	ba

```
1088: GOTO 2;                          /Árvore de decodificação para desvios
  2: R[temp0] <- LSHIFT10(R[ir]);      /Estenda por sinal os 22 bits menores de ir
  3: R[temp0] <- RSHIFT5(R[ir]);       /Deslocando à esquerda 10 bits, e depois à
  4: R[temp0] <- RSHIFT5(R[ir]);       /direita 10 bits. RSHIFT estende o sinal.
  5: R[ir] <- RSHIFT5(R[ir]);          /Move campo COND para IR[13], aplicando
  6: R[ir] <- RSHIFT5(R[ir]);          /RSHIFT5 três vezes. (A extensão do sinal é
  7: R[ir] <- RSHIFT5(R[ir]);          /irrelevante.
  8: IF R[IR[13]] THEN GOTO 12;        /É ba?
     R[IR] <- ADD(R[IR],R[IR]);
  9: IF R[IR[13]] THEN GOTO 13;        /Não é be?
     R[IR] <- ADD(R[IR],R[IR]);
 10: IF Z THEN GOTO 12;                /Execute be
     R[IR] <- ADD(R[IR],R[IR]);
 11: GOTO 2047;                        /Desvio para be não tomado
 12: R[pc] <- ADD(R[pc],R[temp0]);     /Desvio tomado
     GOTO 0;
```

```
13: IF R[IR[13]] THEN GOTO 16;      /É bcs?
     R[IR] <- ADD(R[IR],R[IR]);
14: IF C THEN GOTO 12;              /Execute bcs
15: GOTO 2047;                      /Desvio para bcs não tomado
16: IF R[IR[13]] THEN GOTO 19;      /É bvs?
17: IF N THEN GOTO 12;              /Execute bneg
18: GOTO 2047;                      /Desvio para bneg não tomado
19: IF V THEN GOTO 12;              /Execute bvs
20: GOTO 2047;                     /Desvie para bvs não tomado

2047: R[pc] <- INCPC(R[PC]); GOTO 0; /Incremente %pc e comece novamente
```

Micro-linguagem de Montagem

```
0: R[ir] <- AND(R[pc],R[pc]); READ

100000 0 100000 0 100101 0  1  0 0101 000 00000000000
   A    AMUX   B    BMUX   C   CMUX RD WR  ALU  COND    JUMPADDR
```

AMUX, BMUX = 0: entradas destes registradores vêm do registrador MIR

CMUX = 0: entrada do CMUX vem do registrador MIR

ALU = 0101: operação AND (não afeta códigos de condição)

COND = 000(NEXT): controle passa para a próxima palavra

JUMPADDR não importa (zeros foram colocados arbitrariamente)

Microprograma ARC montado

Microstore Address	A	AMUX	B	BMUX	C	CMUX	RW	ALU	COND	JUMP ADDR
0	10000	0	10000	0	10010	1	0	0101	000	0000000000
1	00000	0	00000	0	00000	0	0	0101	111	0000000000
1152	10010	0	00000	0	00000	1	0	1010	110	1111111111
1280	10000	0	10000	0	00111	0	0	0101	000	0000000000
1281	10010	0	10010	0	10001	0	0	1000	000	0000000000
1282	10001	0	10001	0	10001	0	0	1000	000	0000000000
1283	10000	0	10001	0	10000	0	0	1000	110	0000000000
1600	00000	0	00000	0	00000	0	0	0101	101	1100100010
1601	00000	1	00000	1	00000	1	0	0011	110	1111111111
1602	10010	0	00000	0	10001	0	0	1100	000	0000000000
1603	00000	1	10001	0	00000	1	0	0011	110	1111111111
1604	00000	0	00000	0	00000	0	0	0101	101	1100100010
1605	00000	1	00000	1	00000	1	0	0000	110	1111111111
1606	10010	0	00000	0	10001	0	0	1011	000	0000000000
1607	00000	1	10001	0	00000	1	0	0000	110	1111111111
1608	00000	0	00000	0	00000	0	0	0101	101	1100100101
1609	00000	1	00000	1	00000	1	0	0001	110	1111111111
1610	10010	0	00000	0	10001	0	0	1011	000	0000000000
1611	00000	1	10001	0	00000	1	0	0001	110	1111111111
1624	00000	0	00000	0	00000	0	0	0101	101	1100101101
1625	00000	1	00000	1	00000	1	0	0010	110	1111111111
1626	10010	0	00000	0	10001	0	0	1011	000	0000000000
1627	00000	1	10001	0	00000	1	0	0010	110	1111111111
1688	00000	0	00000	0	00000	0	0	0101	000	1101001101
1689	00000	1	00000	1	00000	1	0	0100	110	1111111111
1690	10010	0	00000	0	10001	0	0	1011	000	0000000000
1691	00000	1	10001	0	00000	1	0	0100	110	1111111111
1760	00000	0	00000	0	00000	0	0	0101	101	1101110001
1761	00000	1	00000	1	00000	0	0	1000	110	0000000000
1762	10010	0	00000	0	10001	0	0	1100	000	0000000000
1763	00000	1	10001	0	10000	0	0	1000	110	0000000000
1792	00000	1	00000	1	10001	0	0	1000	101	1110000010

Principles of Computer Architecture
by M. Murdocca and V. Heuring

Microprograma ARC montado (cont.)

	A	AMUX	B	BMUX	C	CMUX	RW	ALU	COND	JUMP ADDR
1793	10001	0	10001	0	000000	1	10	0101	110	11111111111
1794	10010	0	00000	0	100001	0	00	1100	000	00000000000
1795	00000	1	10001	0	100001	0	00	1000	110	11100000001
1808	00000	1	00000	1	100001	0	00	1000	101	11100010010
1809	10010	0	00000	0	100101	0	00	1111	110	00000101000
40	10010	0	00000	0	100101	0	00	1111	000	00000000000
41	10010	0	00000	0	100101	0	00	1111	000	00000000000
42	10010	0	00000	0	100101	0	00	1111	000	00000000000
43	10010	0	00000	0	100101	0	00	1111	000	00000000000
44	10000	1	00000	0	000000	0	01	0101	110	11111111111
1810	10010	0	00000	0	100001	0	00	1100	000	00000000000
1811	00000	1	10001	0	100001	0	00	1000	110	11100010001
1088	00000	0	00000	0	000000	0	00	0101	110	00000000010
2	10010	0	00000	0	100001	0	00	1010	000	00000000000
3	10000	1	00000	0	100001	0	00	1111	000	00000000000
4	10000	1	00000	0	100001	0	00	1111	000	00000000000
5	10010	0	00000	0	100101	0	00	1111	000	00000000000
6	10010	0	00000	0	100101	0	00	1111	000	00000000000
7	10010	0	00000	0	100101	0	00	1111	000	00000000000
8	10010	0	10010	0	100101	0	00	1000	101	00000000110
9	10010	0	10010	0	100101	0	00	1000	101	00000000110
10	10010	0	10010	0	100101	0	00	1000	010	00000000110
11	00000	0	00000	0	000000	0	00	0101	110	11111111111
12	10000	0	10000	1	010000	0	00	1000	110	00000000000
13	10010	0	10010	1	100101	0	00	1000	101	00000010000
14	00000	0	00000	0	000000	0	00	0101	100	00000000110
15	00000	0	00000	0	000000	0	00	0101	110	11111111111
16	00000	0	00000	0	000000	0	00	0101	101	00000010011
17	00000	0	00000	0	000000	0	00	0101	001	00000000110
18	00000	0	00000	0	000000	0	00	0101	110	11111111111
19	00000	0	00000	0	000000	0	00	0101	011	00000000110
20	00000	0	00000	0	000000	0	00	0101	110	11111111111
2047	10000	0	00000	0	100000	0	00	1110	110	00000000000

Exemplo: Criar a instrução subcc

- **Adicionar a instrução subcc (*subtract*) ao conj. de instruções ARC. subcc usa o formato Aritmético e op3 = 001100.**

```
1584: R[temp0] ← SEXT13(R[ir]);          / Extract rs2 operand
      IF IR[13] THEN GOTO 1586;          / Is second source immediate?
1585: R[temp0] ← R[rs2];                 / Extract sign extended immediate operand
1586: R[temp0] ← NOR(R[temp0], R[0]);    / Form one's complement of subtrahend
1587: R[temp0] ← INC(R[temp0]); GOTO 1603; / Form two's complement of subtrahend
```

	A	AMUX	B	BMUX	C	CMUX	RW	DR	ALU	COND	JUMP ADDR
1584	10010	1	00000	0	10000	1	0	0	1100	101	1100011001 0
1585	00000	0	00000	1	10000	1	0	0	1000	000	0000000000 0
1586	10000	1	00000	0	10000	1	0	0	0111	000	0000000000 0
1587	10000	1	00000	0	10000	1	0	0	1101	110	1100100001 1

Interrupções

- Interrupções por software (ou *traps*)
  - Procedimento chamado pelo hardware qdo condição excepcional
    - Ex. instrução ilegal, overflow, divisão por zero
- Interrupções por hardware (ou interrupções simplesmente)
  - Iniciadas após uma exceção no hardware
    - Ex. aperto de uma tecla no teclado, chegada de uma chamada pelo modem

Tratamento de Interrupções de Software

- Micro-código
  - Procedimento checa os bits de status
  - Desvia para posição inicial do tratador de interrupções (carrega o endereço no PC)
  - Tabela de Desvios
    - Seção fixa da memória
    - Armazena o endereço inicial de cada tratador

Tabela de Desvios

- **Uma tabela de desvios para tratadores de *interrupção por software* (*trap handlers*) e rotinas de serviço de *interrupcões por hardware* (ou interrupções):**

Address	Contents	Trap Handler
	. . .	
60	JUMP TO 2000	Illegal instruction
64	JUMP TO 3000	Overflow
68	JUMP TO 3600	Underflow
72	JUMP TO 5224	Zerodivide
76	JUMP TO 4180	Disk
80	JUMP TO 5364	Printer
84	JUMP TO 5908	TTY
88	JUMP TO 6048	Timer
	. . .	

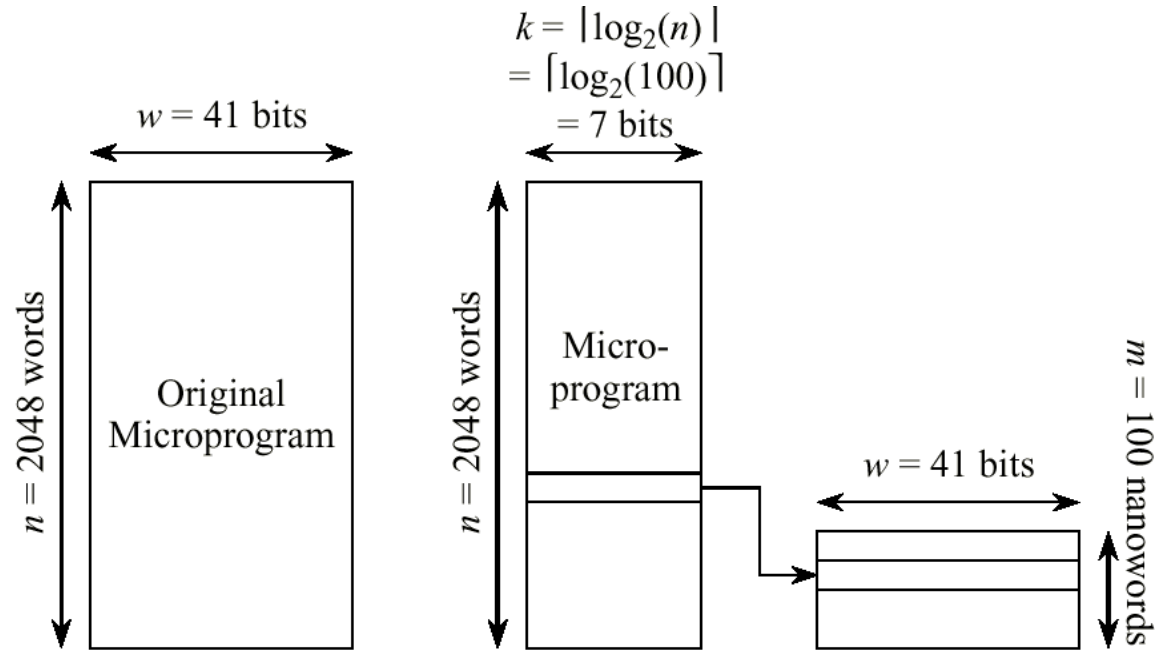
Tratamento de Interrupções de Hardware

- Interrupção assíncrona com o software
  - (pode ocorrer em qualquer ponto do programa)

- Ex. teclado
  - Coloca linha de interrupção em 1
  - CPU liga linha de confirmação quando pronta
    - Arbitragem do barramento
  - Circuito do teclado coloca *vetor de interrupções* no barramento de dados
  - CPU armazena PC e registrador de status na pilha
  - Vetor de interrupções indexa a tabela de desvios, que lista os endereços das rotinas de interrupção

Microprogramação vs. Nanoprogramação

- **(a) Micropro-gramação vs. (b) nano-programação.**



Total Area = $n \times w$ = $2048 \times 41 = 83{,}968$ bits

Microprogram Area = $n \times k = 2048 \times 7$
= 14,336 bits
Nanoprogram Area = $m \times w = 100 \times 41$
= 4100 bits
Total Area = 14,336 + 4100 = 18,436 bits

(a) (b)

Controle Fixo em Hardware

- Outra forma de implementar a Unidade de Controle
  - Flip-flops e portas lógicas em vez de armazenamento de controle e mecanismos de seleção de micro-palavras
  - Passo do microprograma substituídos por estados de uma máquina de estados finitos
- Normalmente utiliza-se uma *linguagem de descrição de hardware* (HDL)
- Linguagem usada no projeto da UC do ARC
  - AHPL (*A Hardware Programming Language*) [Hill e Peterson'87]

Projeto HDL da Unidade de Controle do ARC

- **Cada comando consiste numa parte de dados e numa parte de controle**

```
5: A <- ADD(B,C)                                    ! Parte de dados
       GOTO {10 CONDITIONED ON IR[12]}.             ! Parte de controle
```

- **Rótulo: comando 5**
- **Transferência de dados para o registrador A**
- **Registradores B e C enviados para CLU (Combinational Logic Unit) que efetua a adição**
- **GOTO: Controle transferido para comando 10 se bit 12 do registrador IR = 1; próximo comando caso contrário**

Projeto de Contador Módulo 4 em HDL

- Seqüência de saída 00, 01, 10, 11 repetida continuamente
  - Se linha de entrada x = 1, contador vai para o estado 0

- Seqüência HDL
  - Preâmbulo
  - Comandos numerados
  - Epílogo
    - END SEQUENCE
      - Termina seqüência de comandos
    - END MODULE_NAME
      - Termina o módulo
    - Qualquer coisa entre END SEQ e END MOD ocorre continuamente, independente do comando (estado)

Hardware Description Language

- **Seqüência HDL para um contador módulo-4 reiniciável.**

Preamble

```
MODULE: MOD_4_COUNTER.
INPUTS: x.
OUTPUTS: Z[2].
MEMORY:
```

Statements

```
0: Z ← 0,0;
   GOTO {0 CONDITIONED ON x,
         1 CONDITIONED ON x̄}.
1: Z ← 0,1;
   GOTO {0 CONDITIONED ON x,
         2 CONDITIONED ON x̄}.
2: Z ← 1,0;
   GOTO {0 CONDITIONED ON x,
         3 CONDITIONED ON x̄}.
3: Z ← 1,1;
   GOTO 0.
```

Epilogue

```
END SEQUENCE.
END MOD_4_COUNTER.
```

Derivação do Circuito a partir da HDL

- Parte de Controle
  - Trata sobre como transições são feitas de um comando para o outro
- Parte de Dados
  - Produção de saídas
  - Modificação dos valores de elementos de memória

Derivação da Parte de Controle

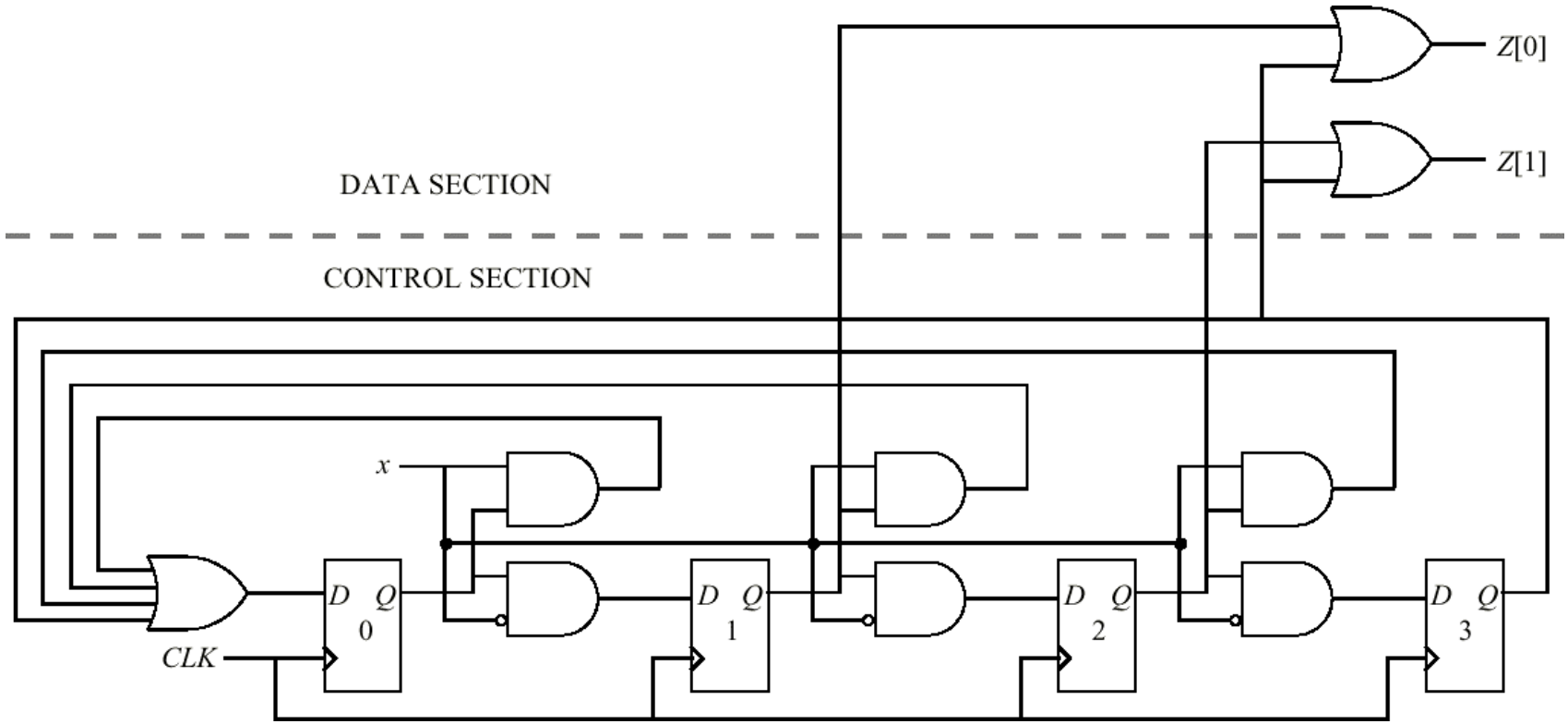
- 4 comandos
  - 1 flip-flop para cada um (*one-hot encoding*)
- Ex.: do comando 0,
  - vai para comando 1 se entrada x = 0
  - vai para comando 0 se entrada x = 1
  - saída do flip-flop 0 entra no flip-flop 1, em AND com x’
  - saída do flip-flop 0 entra no flip-flop 0, em AND com x

Derivação da Parte de Dados

- Produzir o valor correto da saída para cada um dos comandos
- Z[0] é verdadeiro nos comandos 1 e 3
  - OU das saídas dos flip-flops 1 e 3
- Z[1] é verdadeiro nos comandos 2 e 3
  - OU das saídas dos flip-flops 2 e 3

Circuito Derivado da HDL

- **Projeto lógico de um contador módulo-4 descrito em HDL.**



Projeto HDL do ARC

- A Seção de Dados é a mesma, no controle micro-programado ou fixo em hardware

- Seção de Controle
  1. Busque a próxima instrução a ser executada na memória
  2. Decodifique o opcode
  3. Leia os operandos da memória principal ou registradores, se houver
  4. Execute instrução e armazene os dados
  5. Volte ao passo 1

Passagem do Micro-código para HDL

- **Micro-código (busque a próxima instrução)**
  - `0:R[ir] <- AND(R[pc],R[pc]); READ;`
- **HDL**
  - `0:ir <- AND(pc,pc); Read = 1`
- **Micro-código (decodifique)**
  - `1:  DECODE`
- **HDL**
  - `1:  GOTO {2 CONDITIONED ON IR[31]xIR[30],  ! desvio/sethi: op=00`
    `4 CONDITIONED ON IR[31]xIR[30],  ! call: op=01`
    `8 CONDITIONED ON IR[31]xIR[30],  ! aritmético: op=10`
    `10 CONDITIONED ON IR[31]xIR[30]}  ! memória: op=11`

Obs.: x = AND

Descrição HDL da UC do ARC

```
MODULE: ARC_CONTROL_UNIT.

INPUTS:

OUTPUTS: C, N, V, Z.          !setados pela ALU

MEMORY: R[16][32], pc[32], ir[32], temp0[32], temp1[32], temp2[32], temp3[32].

0: ir <- AND(pc,pc); Read <- 1;         ! Busca instrução

   ! Decodifique campo op

1: GOTO {2 CONDITIONED ON IR[31]x!IR[30],  ! Formato desvio/sethi: op=00
         4 CONDITIONED ON IR[31]x!IR[30],  ! Formato call: op = 01
         8 CONDITIONED ON IR[31]x!IR[30],  ! Formato aritmético: op = 10
        10 CONDITIONED ON IR[31]x!IR[30]}. ! Formato de memória: op = 11

   ! Decodifique campo op2

2: GOTO 19 CONDITIONED ON ir[24].         ! Vá para 19 se formato de desvio

3: R[rd] <- ir[imm22]; GOTO 20.           ! sethi
```

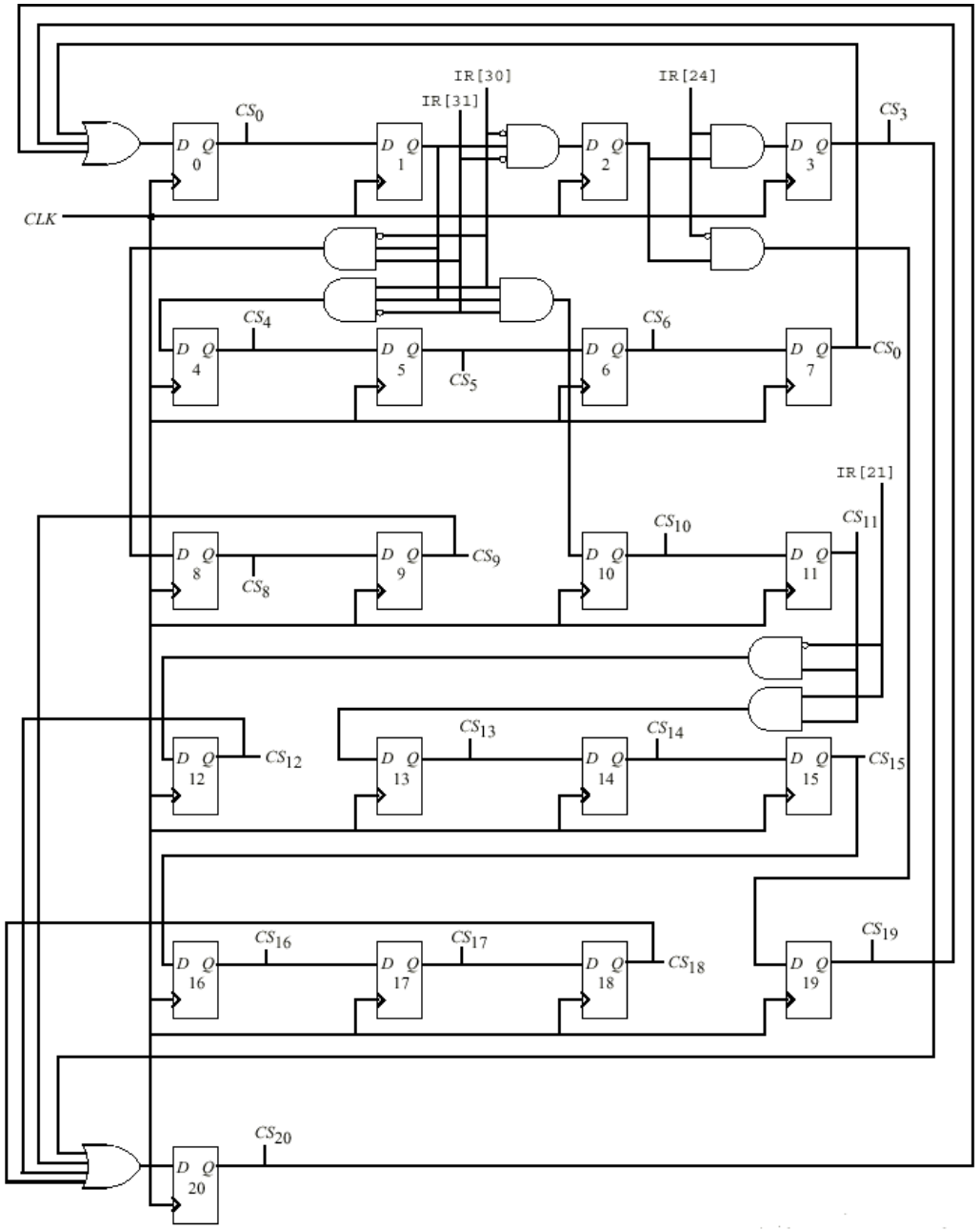
```
4: R[15] <- AND(pc,pc).                    ! call: guarde pc no registrador 15

5: temp0 <- ADD(ir,ir).                    ! Desloque campo disp30 para a esquerda

6: temp0 <- ADD(ir,ir).                    ! Desloque novamente

7: pc <- ADD(pc,temp0); GOTO 0.            ! Desvie para sub-rotina

   ! Leia segundo operando origem para temp0 para formato aritmético

8: temp0 <- { SEXT13(ir) CONDITIONED ON ir[13]xNOR(ir[19:22]),    ! addcc
              R[rs2] CONDITIONED ON ir[13]xNOR(ir[19:22]),       ! addcc
              SIMM13(ir) CONDITIONED ON ir[13]xOR(ir[19:22]),    ! Restante das
              R[rs2] CONDITIONED ON ir[13]xOR(ir[19:22])}        ! instr. Aritméticas

   ! Decodifique campo op3 para formato aritmético

9: R[rd] <- {
   ADDCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010000),      !addcc
   ANDCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010001),      !andcc
   ORCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010010),       !orcc
   NORCC(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 010110),      !orncc
   SRL(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 100110),        !srl
   ADD(R[rs1], temp0) CONDITIONED ON XNOR(IR[19:24], 111000)};       !jmpl

GOTO 20.
```

```
   ! Armazena o segundo operando fonte em temp0, para formato de memória
10: temp0 <- {SEXT13(ir) CONDITIONED ON ir[13],
              R[rs2] CONDITIONED ON ir[13]}.
11: temp0 <- ADD(R[rs1], temp0).        ! Calcula o endereço de memória
    GOTO {12 CONDITIONED ON ir[21],     ! Decodifica op3 para formato de memória
          13 CONDITIONED ON ir[21]}.
12: R[rd] <- AND(temp0, temp0); Read <- 1; GOTO 20.     ! ld
13: ir <- RSHIFT5(ir).                                  ! st
14: ir <- RSHIFT5(ir).
15: ir <- RSHIFT5(ir).
16: ir <- RSHIFT5(ir).
17: ir <- RSHIFT5(ir).
18: r0 <- AND (temp0, R[rs2]); Write <- 1; GOTO 20.
```

```
     ! Instruções de desvio

  19: pc <- { ADD(pc, temp0) CONDITIONED ON ir[28] +                          ! ba
                                            ir[28] x ir[27] x Z +             ! be
                                            ir[28] x ir[27] x ir[26] x C +    ! bcs
                                            ir[28] x ir[27] x ir[26] x ir[25] x N +  ! bneg
                                            ir[28] x ir[27] x ir[26] x ir[25] x V,   ! bvs
              INCPC(pc) CONDITIONED ON ir[28] x ir[27] x Z +
                                            ir[28] x ir[27] x ir[26] x C +
                                            ir[28] x ir[27] x ir[26] x ir[25] x N +
                                            ir[28] x ir[27] x ir[26] x ir[25] x V};

            GOTO 0.

  20: pc <- INCPC(pc); GOTO 0.     ! Incremente o PC, vá para 0 buscar próx. instrução

  END SEQUENCE.

  END ARC_CONTROL_UNIT.
```

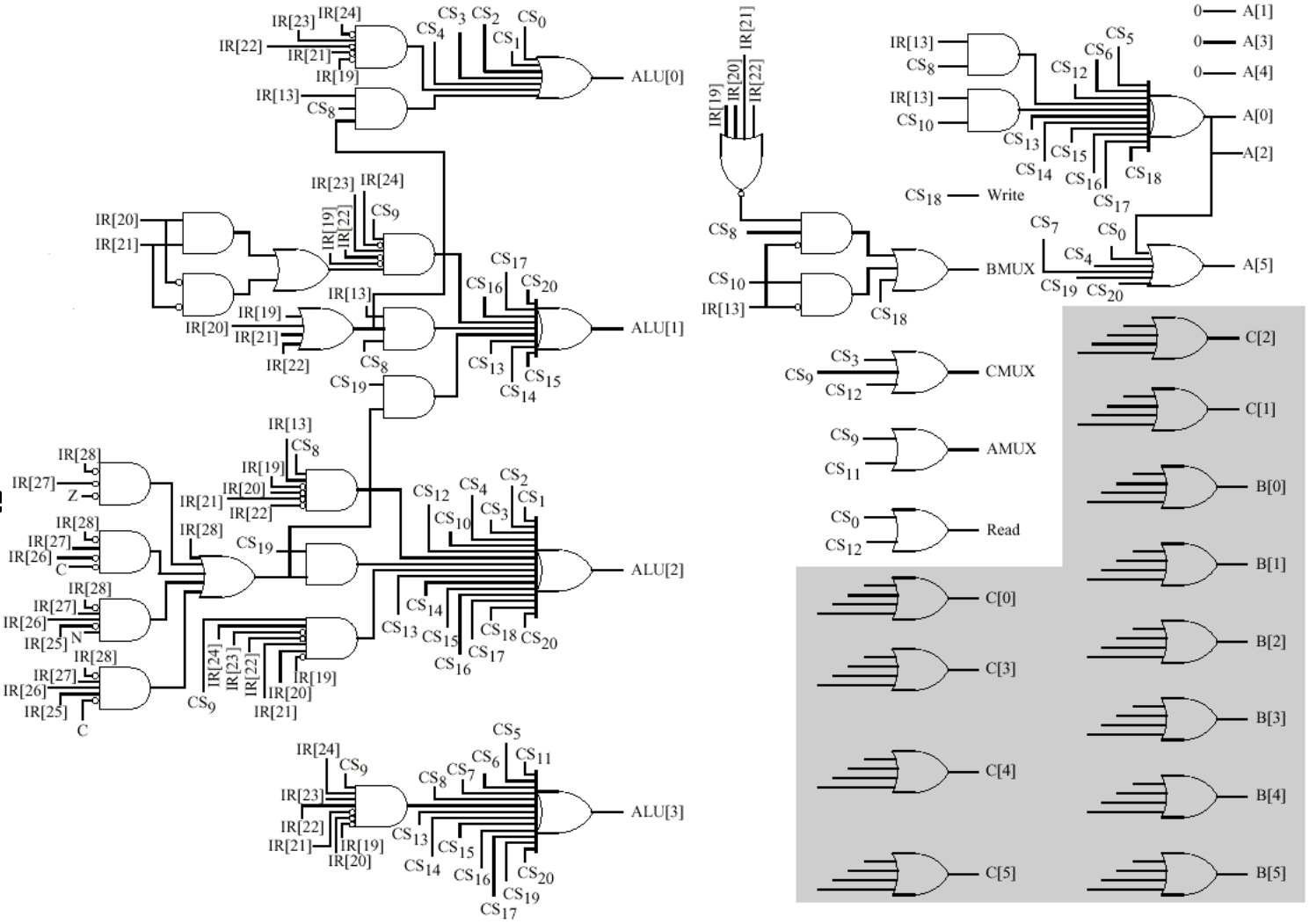
Circuito HDL ARC

- **Unidade de Controle fixo em hardware do ARC: geração dos sinais de controle.**



Circuito HDL ARC (cont.)

- **Seção de controle fixo em hardware do ARC: sinais da parte de dados da unidade de controle para a seção de dados (*datapath*).**

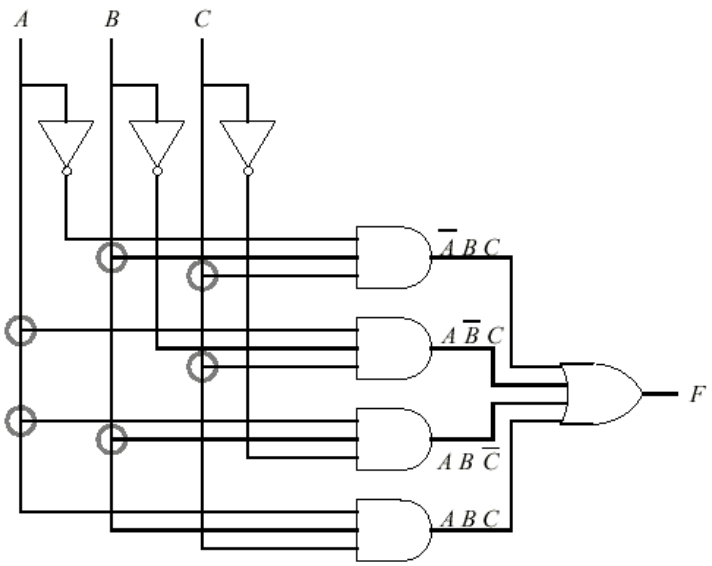


Estudo de Caso: A Linguagem de Descrição de Hardware VHDL

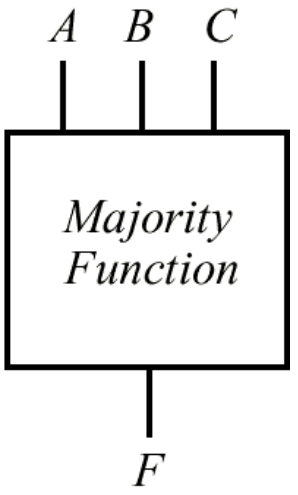
- **A função de maioria. a) tabela verdade, b) implementação AND-OR, c) representação de caixa-preta.**

Minterm Index	*A*	*B*	*C*	*F*
0	0	0	0	0
1	0	0	1	0
2	0	1	0	0
3	0	1	1	1
4	1	0	0	0
5	1	0	1	1
6	1	1	0	1
7	1	1	1	1

a)



b)



c)

Especificação VHDL

Interface specification for the majority component

```
-- Interface
entity MAJORITY is
    port
       (A_IN, B_IN, C_IN: in BIT
           F_OUT: out BIT);
end MAJORITY;
```

Behavioral model for the majority component

```
-- Body
architecture LOGIC_SPEC of MAJORITY is
begin
-- compute the output using a Boolean expression
F_OUT   <=    (not A_IN and B_IN and C_IN) or
              (A_IN and not B_IN and C_IN) or
              (A_IN and B_IN and not C_IN) or
              (A_IN and B_IN and C_IN) after 4 ns;
end LOGIC_SPEC;
```

Especificação VHDL (cont.)

```
-- Package declaration, in library WORK
package LOGIC_GATES is
component AND3
    port (A, B, C : in BIT; X : out BIT);
end component;
component OR4
    port (A, B, C, D : in BIT; X : out BIT);
end component;
component NOT1
    port (A : in BIT; X : out BIT);
end component;
-- Interface
entity MAJORITY is
    port
       (A_IN, B_IN, C_IN: in BIT
       F_OUT: out BIT);
end MAJORITY;
```

Especificação VHDL (cont.)

```
-- Body
-- Uses components declared in package LOGIC_GATES
-- in the WORK library
-- import all the components in WORK.LOGIC_GATES
use WORK.LOGIC_GATES.all
architecture LOGIC_SPEC of MAJORITY is
-- declare signals used internally in MAJORITY
signal A_BAR, B_BAR, C_BAR, I1, I2, I3, I4: BIT;
begin
-- connect the logic gates
NOT_1 : NOT1 port map (A_IN, A_BAR);
NOT_2 : NOT1 port map (B_IN, B_BAR);
NOT_3 : NOT1 port map (C_IN, C_BAR);
AND_1 : AND3 port map (A_BAR, B_IN, C_IN, I1);
AND_2 : AND3 port map (A_IN, B_BAR, C_IN, I2);
AND_3 : AND3 port map (A_IN, B_IN, C_BAR, I3);
AND_4 : AND3 port map (A_IN, B_IN, C_IN, I4);
OR_1 : OR3 port map (I1, I2, I3, I4, F_OUT);
end LOGIC_SPEC;
```