Il controllo specificato come microprogramma

• Nella pratica, set istruzioni più numeroso o istruzioni più complesse (cfr. Intel)

⇒ drastico aumento del numero degli stati

⇒ Impraticabile rappresentare gli stati esplicitamente!

• Idea: molti di questi stati raggiungibile in modo sequenziale

(es. precedente: da stato_0 si passa sempre a stato_1 a prescindere da ingressi)

Cfr. concetti della programmazione

⇒ Rappresentazione dei segnali di controllo al datapath come microistruzioni eseguite in un microprogramma.

Microistruzioni eseguite in sequenza + salti…

⇒ Nel caso di molti stati, porta ad una significativa semplificazione nella specifica del controllo!

• Inoltre, come vedremo questa specifica ha una "naturale" realizzazione che può essere più economica di una FSM in diversi casi

Nota bene:

Abbiamo:

un particolare datapath

ISA (che include il formato delle istruzioni macchina)

Dobbiamo specificare l'unità di controllo!

Solo dopo verrà illustrata la realizzazione.

Passi necessari:

1) **Definire Formato** (rappresentazione simbolica) delle **microistruzioni**:

- Sequenza di campi distinti
- Valori (simbolici) che ogni campo può assumere
- Segnali di controllo associati a ciascun campo

Obiettivi:

- leggibilità del microprogramma

  $\Rightarrow$ ciascun campo associato ad una funzione specifica

  [es. 1 campo per operaz. ALU, 3 per scelta sorgenti e destinazione]

- impedire scrittura microistruzioni incoerenti / istruzioni non supportate

  $\Rightarrow$ campi diversi per insiemi disgiunti di segnali di controllo;

NB: Segnali non usati in modo completamente indipendente possono essere associati allo stesso campo (per minimizzare lunghezza delle microistruzioni), mentre gruppi di segnali "indipendenti" vanno associati a campi distinti

Nome del campo	Valore del campo	Funzione del campo con tale valore
Label	Una stringa qualunque	Questo campo è utilizzato per specificare le etichette utilizzate per controllare la sequenza delle microistruzioni. Le etichette che terminano con 1 o 2 sono utilizzate durante gli smistamenti tramite tabelle di salto indicizzate sulla base del codice operativo; le altre etichette sono utilizzate come destinazioni dirette per i salti tra microistruzioni. Le etichette non generano direttamente segnali di controllo, ma sono utilizzate per definire il contenuto delle tabelle di smistamento e per generare i segnali di controllo relativi al campo Sequencing.
ALU control	`Add`	Indica alla ALU di sommare.
	`Subt`	Indica alla ALU di sottrarre; è utilizzato nell'implementazione del confronto per i salti condizionati.
	`Func code`	Utilizza il campo funct dell'istruzione per determinare i segnali di controllo della ALU.
SRC1	`PC`	PC è il primo ingresso della ALU.
	`A`	Il registro A è il primo ingresso della ALU.
SRC2	`B`	Il registro B è il secondo ingresso della ALU.
	`4`	Il valore 4 va sul secondo ingresso della ALU.
	`Extend`	L'uscita dell'unità di estensione del segno va sul secondo ingresso della ALU.
	`Extshft`	L'uscita dell'unità di scalamento va sul secondo ingresso della ALU.
Register control	`Read`	Legge due registri utilizzando i campi rs e rt di IR come numeri di registro, ponendo i dati letti nei registri A e B.
	`Write ALU`	Scrive nel register file utilizzando il campo rt di IR come numero di registro ed il contenuto di ALUOut come dato.
	`Write MDR`	Scrive nel register file utilizzando il campo rt di IR come numero di registro ed il contenuto di MDR come dato.
Memory	`Read PC`	Legge la memoria utilizzando PC come indirizzo; il risultato è scritto in IR (e nel MDR).
	`Read ALU`	Legge la memoria utilizzando ALUOut come indirizzo; il risultato è scritto in MDR.
	`Write ALU`	Scrive nella memoria utilizzando ALUOut come indirizzo; il dato è contenuto in B.
PCWrite control	`ALU`	Scrive l'uscita della ALU in PC
	`ALUOut-cond`	Se l'uscita Zero della ALU è attiva, scrive in PC il contenuto del registro ALUOut.
	`Jump address`	Scrive in PC l'indirizzo di salto ricavato dall'istruzione.
Sequencing	`Seq`	Seleziona sequenzialmente la microistruzione successiva.
	`Fetch`	Torna alla prima microistruzione, che inizia l'esecuzione di una nuova istruzione.
	`Dispatch i`	Esegue uno smistamento utilizzando la ROM specificata da `i` (1 o 2).

Notare che:

• In *Register control* sono previsti due possibili valori per la scrittura dei registri:

- Write ALU: scrive nel registro *rd* il contenuto di *ALUout*
  [usato nelle istruzioni di Tipo-R]
- Write MDR: scrive nel registro *rt* il contenuto di *MDR*
  [usato nell'istruzione lw]

Sono quindi escluse le combinazioni:

- scrittura in *rt* del contenuto di *ALUout*
- scrittura in *rd* del contenuto di *MDR*

possibili con il datapath progettato ma non usate in istruzioni MIPS considerate.

• In *Memory* i valori previsti non permettono ad esempio di:

- leggere la memoria indirizzata da *ALUout* e scrivere il valore letto in *IR*
- scrivere in memoria (il dato contenuto in *B*) nella locazione indirizzata da PC

A ciascun campo (a parte Sequencing) possiamo già associare segnali di controllo, il cui valore è determinato dal valore simbolico del campo stesso
(cfr. Assembler vs. Linguaggio Macchina)

Nome del campo	Valore	Segnali attivi	Commenti
Controllo della ALU	Add	ALUOp=00	Forza la ALU ad eseguire la somma.
	Subt	ALUOp=01	Forza la ALU ad eseguire la sottrazione; si implementa così anche il confronto per i salti.
	Codice func	ALUOp=10	Usa il codice funzione dell'istruzione per determinare il controllo della ALU.
SRC1	PC	ALUSrcA=0	Usa il PC come primo ingresso della ALU.
	A	ALUSrcA=1	Il registro A è il primo ingresso della ALU.
SRC2	B	ALUSrcB=00	Il registro B è il secondo ingresso della ALU.
	4	ALUSrcB=01	Si usa 4 come secondo ingresso della ALU.
	Extend	ALUSrcB=10	Si usa l'uscita dell'unità di estensione del segno come secondo ingresso della ALU.
	Extshft	ALUSrcB=11	Si usa l'uscita dell'unità di scalamento di due posizioni come secondo ingresso della ALU.
Controllo del registro	Lettura		Si leggono due registri usando i campi rs e rt di IR come numeri dei registri e si mette il dato nei registri A e B.
	Scrittura ALU	RegWrite, RegDst=1, MemtoReg=0	Si scrive un registro usando il campo rd di IR come numero del registro e il contenuto di ALUOut come dato.
	Scrittura MDR	RegWrite, RegDst=0, MemtoReg=1	Si scrive un registro usando il campo rt di IR come numero del registro e il contenuto di MDR come dato.
Memoria	Lettura PC	MemRead, IorD=0 IRWrite	Si legge la memoria usando PC come indirizzo; si scrive il risultato in IR (e in MDR).
	Lettura ALU	MemRead, IorD=1	Si legge la memoria usando ALUOut come indirizzo; si scrive il risultato in MDR.
	Scrittura ALU	MemWrite, IorD=1	Si scrive in memoria usando ALUOut come indirizzo, e il contenuto di B come dato.
Controllo della scrittura di PC	ALU	PCSource=00, PCWrite	Si scrive l'uscita della ALU in PC.
	ALUOut-cond	PCSource=01, PCWriteCond	Se è attiva l'uscita Zero della ALU si scrive in PC il contenuto del registro ALUOut.
	Indirizzo di salto	PCSource=10, PCWrite	Si scrive in PC l'indirizzo di salto contenuto nell'istruzione.

Comandi non specificati:

- a elementi di memoria (lettura e scrittura):
  = 0
- a MUX e controllo elementi combinatori (ALU):
  = don't care

2) **Creare Microprogramma**: rappresentazione simbolica del controllo traducibile automaticamente in circuiti logici di controllo.

Possibilità di verifica consistenza interna (non contraddittorietà)
ed esterna (rispetto ai requisiti).

Nel caso MIPS il microprogramma corrisponde esattamente al diagramma degli stati individuato nella specifica del processore come FSM.

NB: Ciò fa già capire come la realizzazione possa essere effettuata mediante:

- sequenzializzatore
- macchina a stati finiti!

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

Note:

Nel microprogramma, le microistruzioni sono disposte in sequenza.

Ciascuna istruzione può essere etichettata da una label (campo *Label*)
che serve ad identificarla nei salti.

Scelta della microistruzione successiva dipende da campo *Sequencing*:
- Seq: eseguita l'istruzione successiva nel microprogramma
- Fetch: eseguita la microistruzione di fetch, etichettata con label Fetch,
  che di fatto corrisponde all'esecuzione di una nuova istruzione MIPS
- Dispatch i: l'istruzione successiva viene scelta, sulla base degli ingressi
  (campo opcode di IR), entro la tabella di smistamento i.

Ciascuna tabella contiene la corrispondenza
ingresso → label della istruzione cui saltare

Le label nel campo *Label* terminano con un numero *i* indicante
la tabella di smistamento in cui sono contenute (vedi microprogramma).

Oltre al microprogramma, dobbiamo definire le tabelle di smistamento (forma simbolica)
- sulla base di ciascun possibile valore di opcode, a quale istruzione si salta -

Tabella di smistamento del microcodice 1

Campo codice operativo	Nome del codice operativo	Valore
000000	Formato-R	Rformat1
000010	jmp	JUMP1
000100	beq	BEQ1
100011	lw	Mem1
101011	sw	Mem1

Tabella di smistamento del microcodice 2

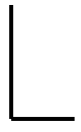
Campo codice operativo	Nome del codice operativo	Valore
100011	lw	LW2
101011	sw	SW2

Con ciò la specifica è completa:
- formato microistruzioni
- associazione valori campi di controllo - segnali controllo a datapath } Cfr. Ling. Macchina
- microprogramma
- tabelle di smistamento } Cfr. Ling. Assembler

REALIZZAZIONE DELLA SPECIFICA CON MICROPROGRAMMA

• Dobbiamo decidere come realizzare:
  - funzione di sequenzializzazione (determina la sequenza di computazione)
  - funzione di controllo (determina le uscite di controllo Data Path)

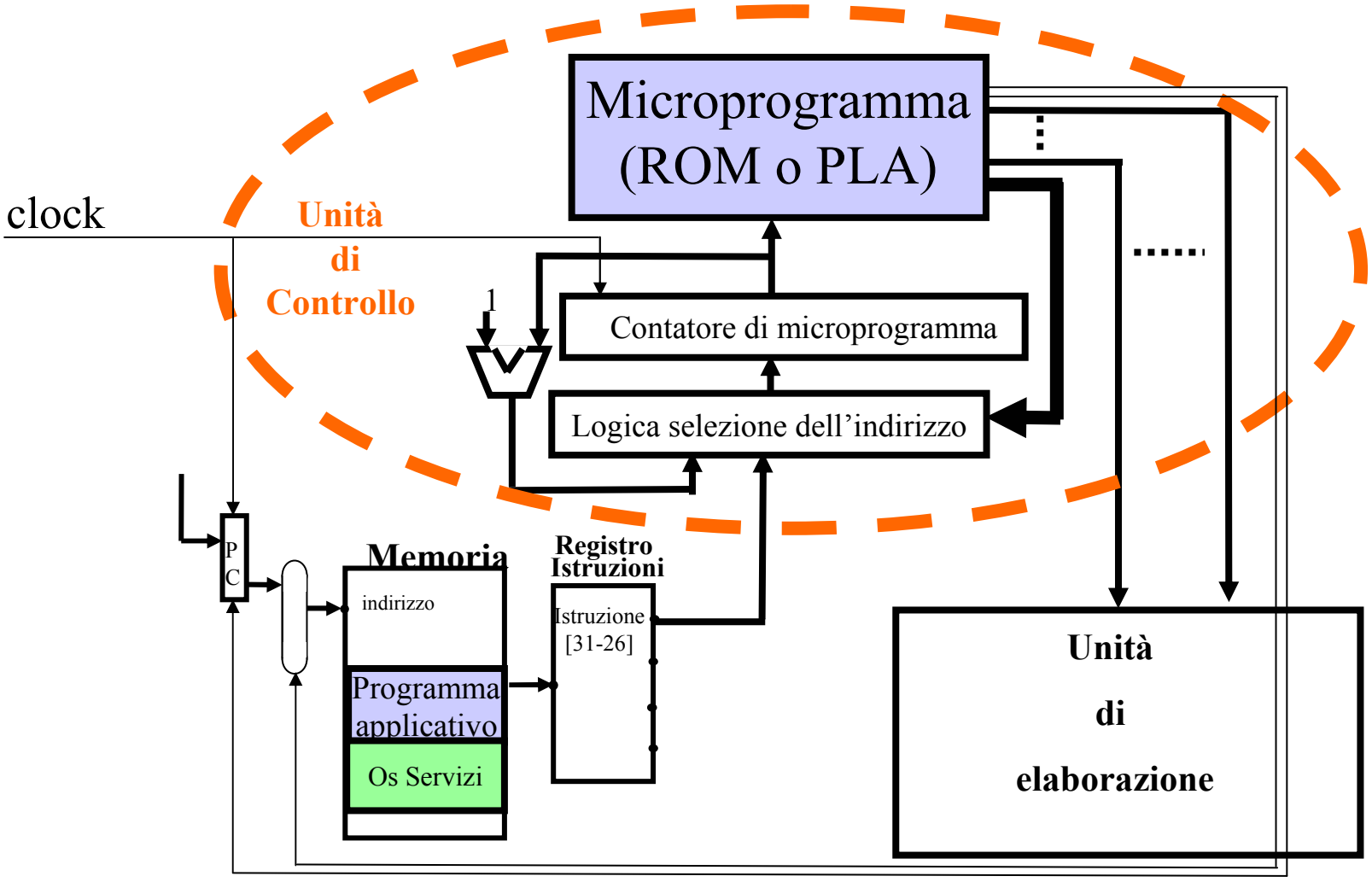
• Esistono due modalità di realizzazione:
  - come macchina a stati finiti:
    Funzione di controllo = funzione di uscita [con PLA o ROM]
    Funzione di sequenzializzazione = funzione di stato futuro [con PLA o ROM]
    $\Rightarrow$ funzione stato futuro codifica esplicitamente lo stato
    NB: vedremo brevemente questa tecnica alla fine…
  - Usando una ROM per memorizzare la funzione di controllo ed un contatore che fornisce in maniera sequenziale lo stato futuro (indirizzo microistruzione): si elimina la codifica esplicita dello stato futuro all'interno dell'unità di controllo



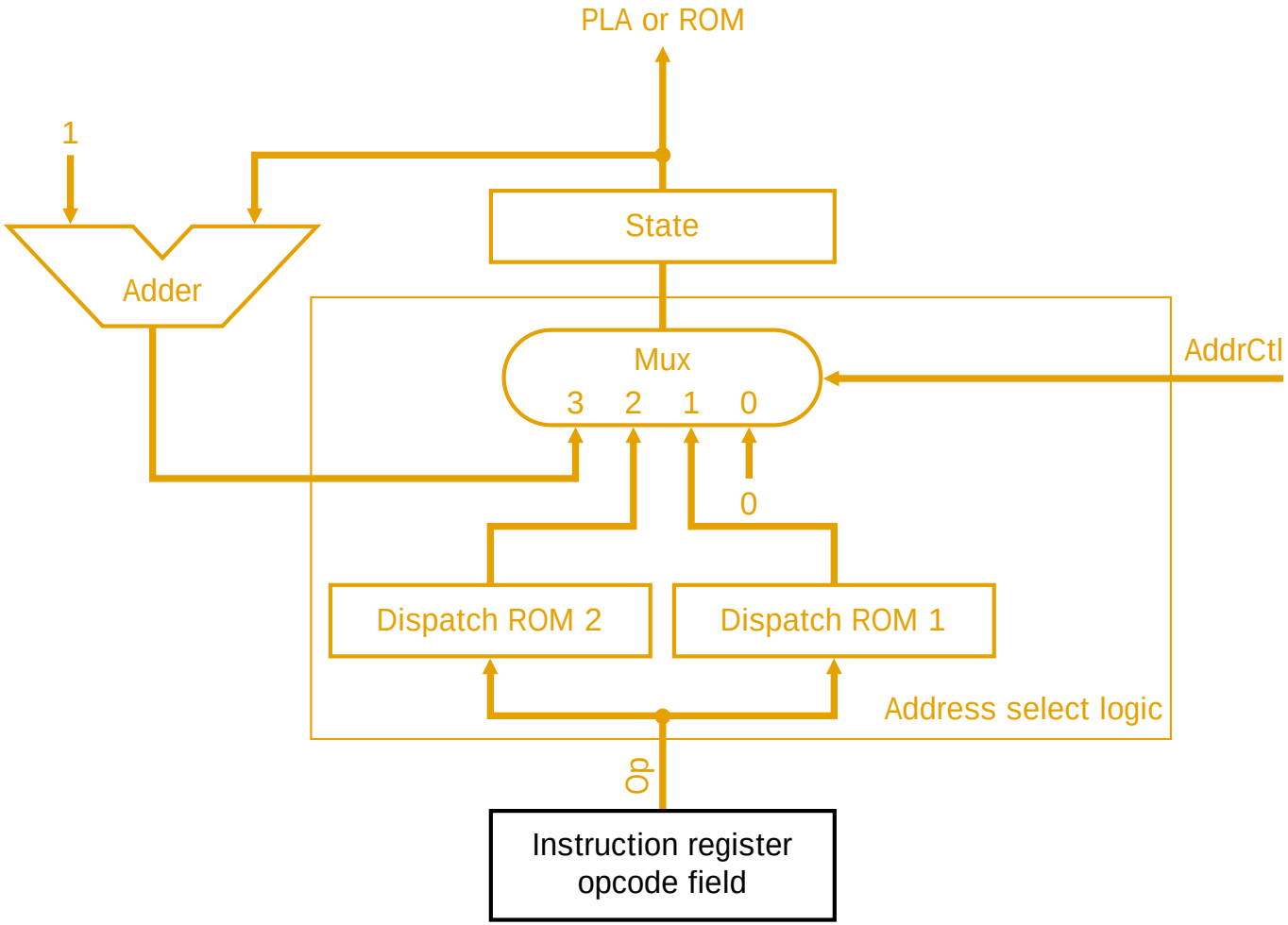
Controllo microprogrammato

Controllo Microprogrammato

Durante l'esecuzione di un programma applicativo i circuiti interpretano iterativamente il microprogramma sulle istruzioni del < programma applicativo ∘ i servizi OS> .



- Logica di selezione dell'indirizzo, sulla base del segnale di controllo proveniente da microistruzione corrente, sceglie l'indirizzo successivo tra:
  - stato (indirizzo) incrementato
  - stato 0 (di fetch)
  - stato determinato da tabelle di dispatch funzione di opcode
- Segnale di controllo AddrCtl a due bit (proveniente da microistruzione).

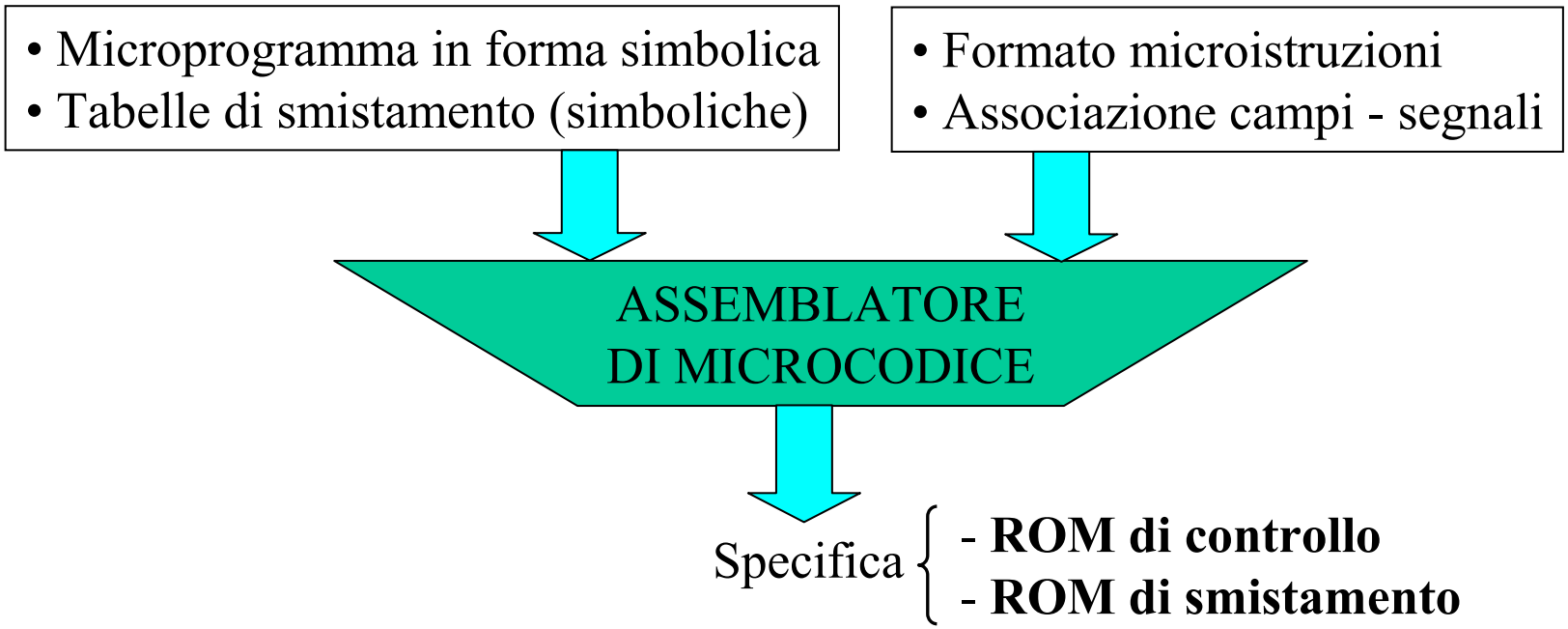


A questo punto possiamo associare i valori simbolici del campo Sequencing al segnale di controllo *AddCtl*:

- *Seq* *AddCtl* = 11
- *Fetch* *AddCtl* = 00
- *Dispatch 1* *AddCtl* = 01
- *Dispatch 2* *AddCtl* = 10

$\Rightarrow$ Abbiamo quindi la corrispondenza completa valori campi-segnali

Fase successiva: corrisponde alla "compilazione" del Linguaggio Assembler in LM



1) Traduzione dei campi di ogni microistruzione nei segnali corrisponenti:
da microprogramma + tabella di associazione campi-segnali

2) Assegnare indirizzi alle microistruzioni [mantenendo sequenzialità]

⇨ Specifica **ROM di controllo**: ogni parola corrisponde a una microistruzione

Mantenendo l'ordine nel microprogramma (coincidente con codifica stati FSM):

Numero dello stato	Bit 17-2 della parola di controllo	Bit 1-0 della parola di controllo
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000000	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

Bit 17-2: bit di controllo (NB: corrispondenti a ciascuno stato del diagramma)
Bit 1-0: bit di controllo indirizzo: associati a campo *Sequencing* della microistruzione

3) Specifica delle **ROM di Smistamento**

da tabelle di smistamento simboliche

\+ indirizzi assegnati alle istruzioni (Tabella label - indirizzo)

ROM di smistamento 1

Op	Nome del codice operativo	Valore
000000	Formato-R	0110
000010	jmp	1001
000100	beq	1000
100011	lw	0010
101011	sw	0010

ROM di smistamento 2

Op	Nome del codice operativo	Valore
100011	lw	0011
101011	sw	0101

NB: gli altri elementi delle ROM [ciascuna con 2^6 * 4 bit] sono vuote in questo caso [perché si sono considerate solo poche istruzioni]

Controllo microprogrammato (realizzato con sequenzializzatore): valutazione

- Specifica come macchina a stati finiti: abbiamo visto che, nel caso delle 2 ROM:

$$\underbrace{2^4 * 16}_{\text{uscita}} + \underbrace{2^{10} * 4}_{\text{stato futuro}} = \underbrace{256}_{6\%} + \underbrace{4096}_{94\%}$$

- Istruzioni complesse (p.es. virgola mobile) sono caratterizzate da sequenze di stati indipendenti da ingressi (opcode); all'aumentare del set di istruzioni disponibili, queste sequenze diventano via via più frequenti
- Inoltre, complessità e numerosità delle istruzioni porta ad aumento numero stati totali

  ⇒ In una macchina a stati, la funzione di stato futuro deve comunque codificare tutte le transizioni stato-stato: anche per quelle indipendenti da ingressi lo stato destinazione deve essere esplicitamente codificato;
  in una tabella di verità, vi sarà una riga per ogni transizione [opcode don't care].

  Nel controllo microprogrammato, la funzione di controllo ha solo due bit in più per decidere la modalità di selezione: transizioni "sequenziali" non codificano esplicitamente lo stato destinazione, ma solo che bisogna scegliere il "seguente". Le ROM di Smistamento memorizzano solo stati cui si arriva in dipendenza dall' ingresso!

  ⇒Stati in sequenza non codificati, semplificando unità di controllo wrt. FSM

VEDIAMO IN DETTAGLIO…

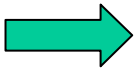
Implementazione di controllo e smistamento tramite ROM: dimensioni

ROM di controllo: 10 microistruzioni $\Rightarrow$ 4 bit di indirizzo
ciascuna parola (microistruzione) 16+2 = 18 bit

Dimensione: $2^4 * 18 = 288$ bit

ROM di smistamento: indirizzate da Opcode $\Rightarrow$ 6 bit di indirizzo
ciascuna parola ha 4 bit

Dimensione: $2^6 * 4 = 256$ bit



Dimensione totale = 288 + 2*256 = 800 bit

Implementazione di controllo e smistamento tramite PLA: dimensioni

PLA di controllo: funzione stato (4 bit) $\Rightarrow$ 16 + 2 bit di controllo di indirizzo

Avevamo visto che per 16 uscite si usano 10 mintermini (tutti i 10 stati); quindi i 2 bit di controllo di indirizzo useranno parte di questi mintermini

Dimensione: = 4*10 + 10*18 = 220

PLA di smistamento 1: funzione bit di opcode (6 bit) $\Rightarrow$ 4 bit di indirizzo [stato]
dalla specifica ROM di smistamento 1 si vede che ci sono 5 combinazioni di opcode che interessano: 5 mintermini

Dimensione: = 6*5 + 5*4 = 50

PLA di smistamento 2: funzione bit di opcode (6 bit) $\Rightarrow$ 4 bit di indirizzo [stato]
da specifica ROM di smistamento 2 si vede che ci sono 2 combinazioni di opcode che interessano: 2 mintermini

Dimensione: = 6*2 + 2*4 = 20

$\Rightarrow$ Dimensione totale = 220 + 50 + 20 = 290

Confronto con implementazione “a stato esplicito”

Tramite ROM:

	Controllo	Stato futuro/ smistamento	Totale
Stato esplicito	256	4096	4.3K
Microprog.	288	512	800

Tramite PLA:

	Controllo	Stato futuro/ smistamento	Totale
Stato esplicito	200	140	340
Microprog.	220	70	290

⇒ Lieve aumento della dimensione per il controllo (dovuta all’introduzione dei segnali AddrCt per la scelta della microistruzione successiva) compensato dalla diminuzione della dimensione per stato futuro

Ottimizzazioni della logica di controllo

Favorite dall'uso di strumenti CAD ma anche da opportune scelte di progettazione:

- Minimizzazione logica delle funzioni di controllo e smistamento:
  - sfruttamento dei termini don't care
  - individuazione di segnali di ingresso che identificano istruzioni [nell'esempio lw e sw sono gli unici ad avere Op5=1] semplificando ROM/PLA di smistamento: favorita da scelta progettuale che assegna codici operativi correlati ad istruzioni che condividono stessi stati (stesse microistruzioni)

- Assegnazione degli stati:
  stati (indirizzi delle microistruzioni) scelti in modo da semplificare equazioni che legano gli stati alle uscite di controllo
  [l'assemblatore di microcodice può assegnare indirizzi in modo opportuno nel rispetto dei vincoli di sequenzialità delle microistruzioni]

• Riduzione della memoria di controllo mediante tecniche di *codifica*:
riduzione ampiezza delle microistruzioni:
- Gruppi di linee nelle microistruzioni possono essere codificate
in un minor numero di bit.
Es. solo uno tra i 6 bit 13-8 della parola di controllo è attivo: codifica in 3 bit

Numero dello stato	Bit 17-2 della parola di controllo	Bit 1-0 della parola di controllo
0	1001010000001000	11
1	0000000000011000	01
2	0000000000010100	10
3	0011000000000000	11
4	0000001000000010	00
5	0010100000000000	00
6	0000000001000000	11
7	0000000000000011	00
8	0100000010100100	00
9	1000000100000000	00

NB: tempo di decodifica di solito trascurabile, costo HW decoder compensato da riduzione costo memoria di controllo

- uso di un campo *formato* delle microistruzioni (formati diversi per microistruzioni!)
[fornisce valori di default per segnali di controllo non specificati]
p. es. formato per microistruzioni di accesso in memoria e
formato per operaz. ALU (specifica implicitamente no read/write memoria)

In generale:

- Microcodice orizzontale (minimamente codificato)
  - più flessibile (può specificare qualunque combinazione di segnali)
  - aumenta costo dell'HW (memoria di controllo più ampia)
  - favorisce buoni T_{clock} e CPI [ma attenzione a memoria di controllo!]

VS.

- Microcodice verticale (massimamente codificato)
  - minor costo HW (costo decodificatori < riduzione costo memoria)
  - peggiori prestazioni (tempo di decodifica che può far aumentare T_{clock} + riduzione combinazioni possibili di segnali nella stessa microistruzione che può richiedere un maggior CPI)

NB: ABBIAMO VISTO

- Processore multi-ciclo [Unità di controllo sequenziale]
  - **specifica come FSM** e **implementazione come FSM**
  - **specifica come microprogramma** e **implementazione con sequenzializzatore**

Sono possibili anche gli altri due passaggi!

- **specifica come microprogramma** e **implementazione come FSM**

Ogni istruzione corrisponde ad uno stato: codifica (numerica) dello stato

$\Rightarrow$ dai valori dei campi di controllo delle microistruzioni
deriva la funzione:

stato $\rightarrow$ uscita [funzione di uscita]

$\Rightarrow$ dal campo *Sequencing* di una microistruzione (stato) + tabelle smistamento simboliche [che codificano istruzione futura sulla base di ingressi opcode]
deriva la funzione:

stato*ingressi $\rightarrow$ stato futuro [funzione di stato futuro]

• **specifica come FSM** e **implementazione con sequenzializzatore**

Ogni stato corrisponde ad una microistruzione: assegnazione di un ordine agli stati
(conviene: transizioni tra stati indipendenti dall'ingresso ⇒ stati posti in sequenza!)

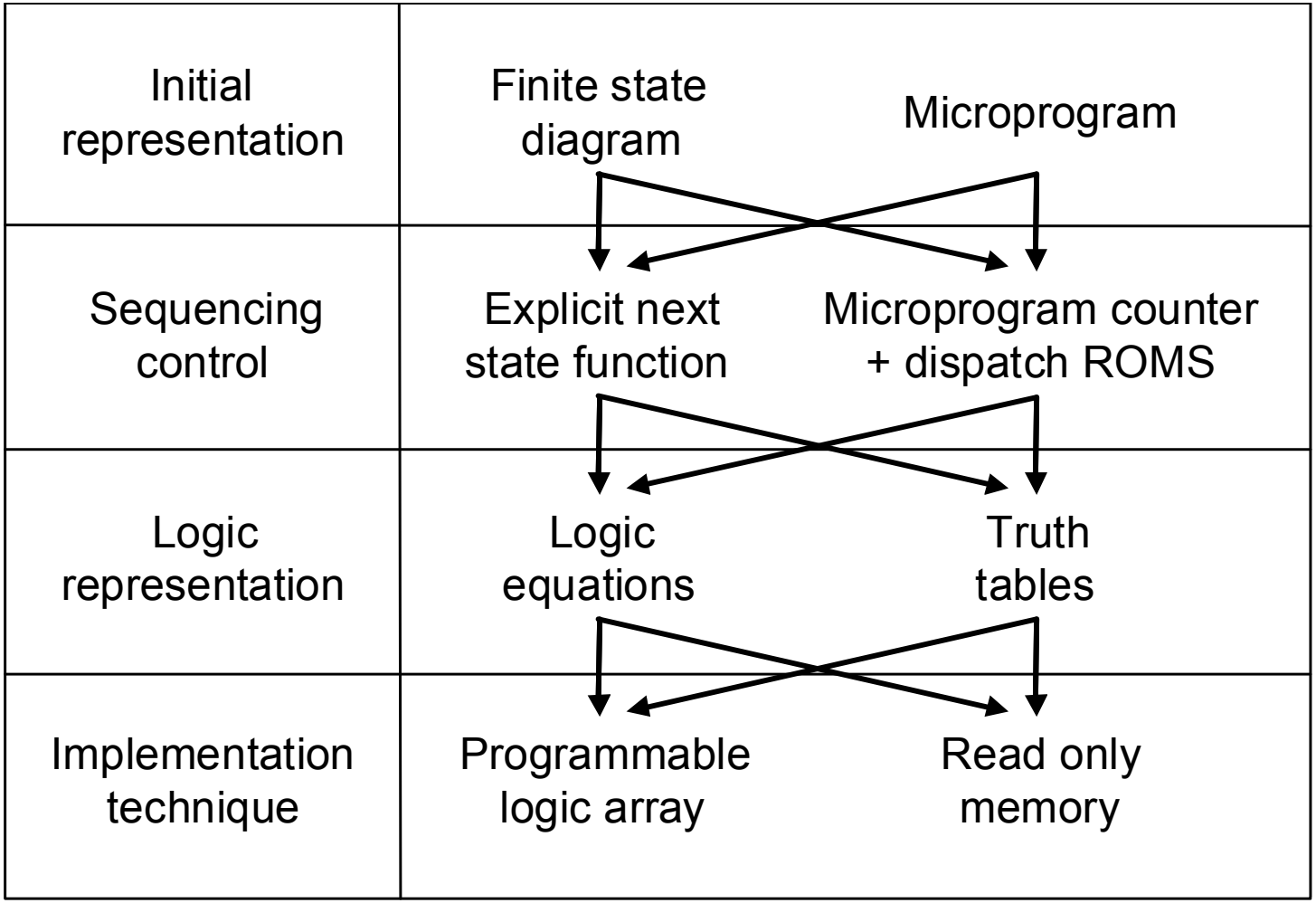
⇒ dalla funzione di uscita derivano per ogni microistruzione
i segnali di controllo da attivare

⇒ dalla funzione di stato futuro si può stabilire se vi sia un'unica transizione
verso la microistruzione seguente, verso la microistruzione di fetch o se
la transizione dipenda dall'ingresso [tabella di smistamento], ovvero:

campo *Sequencing*
tabelle di smistamento

NB: l'uso di strumenti automatici per la realizzazione (in grado di applicare tecniche di ottimizzazione) rende le fasi di specifica e realizzazione relativamente indipendenti! Si può scegliere la specifica che risulta più semplice e "gestibile"

Controllo di un processore-multiciclo: Riepilogo specifica e realizzazione



- Modalità di rappresentazione:

  scelte per ragioni di semplicità progettuale

- Realizzazione di stato futuro:

  sequenzializzatore più efficiente all'aumentare del numero di stati

  e delle sequenze composte da stati consecutivi senza salti

- Realizzazione logica di controllo [parti combinatorie]:

  PLA sono più efficienti (vantaggio non sussiste se ROM è densa)

  ROM è soluzione più adeguata se funzione di controllo è in un chip diverso rispetto a unità di elaborazione (possibilità di cambiare microcodice indipendentemente dal resto del processore)

  └— NB: di fatto per ragioni di efficienza l'unità di controllo è oggi sempre implementata nello stesso chip. Inoltre, con CAD difficoltà di progettazione PLA non sussiste più.

  NB2: CACHE $\Rightarrow$ ROM di microcodice non molto più veloce di RAM con LM: se microprogramma non è efficiente, sequenze di istruzioni macchina possono essere più efficienti di una corrispondente istruzione complessa!