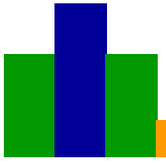


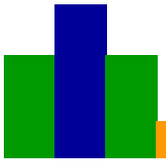
Lecţia 8:

Procesorul:
Calea de date si controlul - II

G Stefănescu — Universitatea Bucureşti

Arhitectura sistemelor de calcul, Sem.1
Octombrie 2016—Februarie 2017

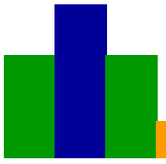
După: D. Patterson and J. Hennessy, Computer Organisation and Design



Procesorul: Calea de date si controlul

Cuprins:

- Generalitati
- Calea de date
- O prima implementare
- *Implementare cu cicluri multiple*
- Microprogramare
- Exceptii
- Concluzii, diverse, etc.



Performanta implementarii cu un ciclu

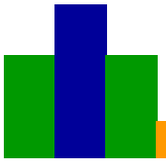
Performanta implementarii: Implementarea cu un ciclu este

- *corectă*
- ... dar *neutilizată* în procesoarele moderne

Motiv: lungimea ciclului ar fi prea mare

- cea mai lungă cale este pentru *load*:
  - 5 faze: ALU şi 4 accesări de memorii - IM (instrucţiuni), RF (regiştri), DM (date), RF (regiştri, din nou).

Există posibilitatea de a folosi un ciclu de lungime variabilă pentru fiecare instrucţiune, dar câştigul este nesemnificativ...

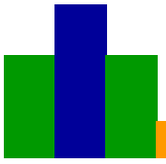


..Performanta implementarii cu un ciclu

Exemplu: Comparăm ciclul de lungime fixă cu cel de lungime variabilă când timpii sunt:

- acces memeorie (IM, DM)- 2 ns;
  ALU ori sumator - 2ns;
  acces regiştri (RF) - 1ns;
- multiplexori, circuite de control, access PC, Sign Extend, conexiuni - 0 ns.

Presupunem distribuţia: 24% - load; 12% - store, 44% ALU, 18% - branch, 2% jump.



..Performanta implementarii cu un ciclu

Durata instrucţiunilor:

Operaţii ALU: 2(IM)+1(RF)+2(ALU)+0(DM)+1(RF)=6ns
Load: 2(IM)+1(RF)+2(ALU)+2(DM)+1(RF)=8ns
Store: 2(IM)+1(RF)+2(ALU)+2(DM)+0(RF)=7ns
Branch: 2(IM)+1(RF)+2(ALU)+0(DM)+0(RF)=5ns
Jump: 2(IM)+0(RF)+0(ALU)+0(DM)+0(RF)=2ns

Cu ciclu de lungime fixă:

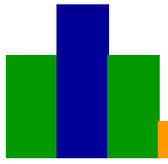
$$\texttt{CPU}_{\texttt{time}} = 8\texttt{ns}$$

Cu ciclu de lungime variabilă:

$$\begin{aligned}\texttt{CPU}_{\texttt{time}} &= 8 \times 0.24 + 7 \times 0.12 + 6 \times 0.44 + 5 \times 0.18 + 2 \times 0.02 \\ &= 6.3\texttt{ns}\end{aligned}$$

Performanţa obţinută $\texttt{Perf var}/\texttt{Perf fix}$ este:

$$\texttt{CPU}_{\texttt{time}}\ \texttt{fix}/\texttt{CPU}_{\texttt{time}}\ \texttt{var} = 8/6.3 = 1.27$$



Implementare cu cicluri multiple

Implementare cu cicluri multiple:

- In implementarea cu *cicluri multiple*, fiecare *pas* (ori *fază*) va dura *1 ciclu*.
- Avantaj: Unele *componente* funcţionale pot fi *utilizate de mai multe ori* la o singură instrucţiune.
- Schemă de ansamblu:
  - Folosim o *unică memorie* pentru *instrucţiuni* şi *date*.
  - Folosim un *unic ALU* (care *ia şi locul sumatoarelor*).
  - Se folosesc *regiştri auxiliari* pentru a *reţine datele* de la un ciclu la altul când procesăm o instrucţiune.

..Implementare cu cicluri multiple

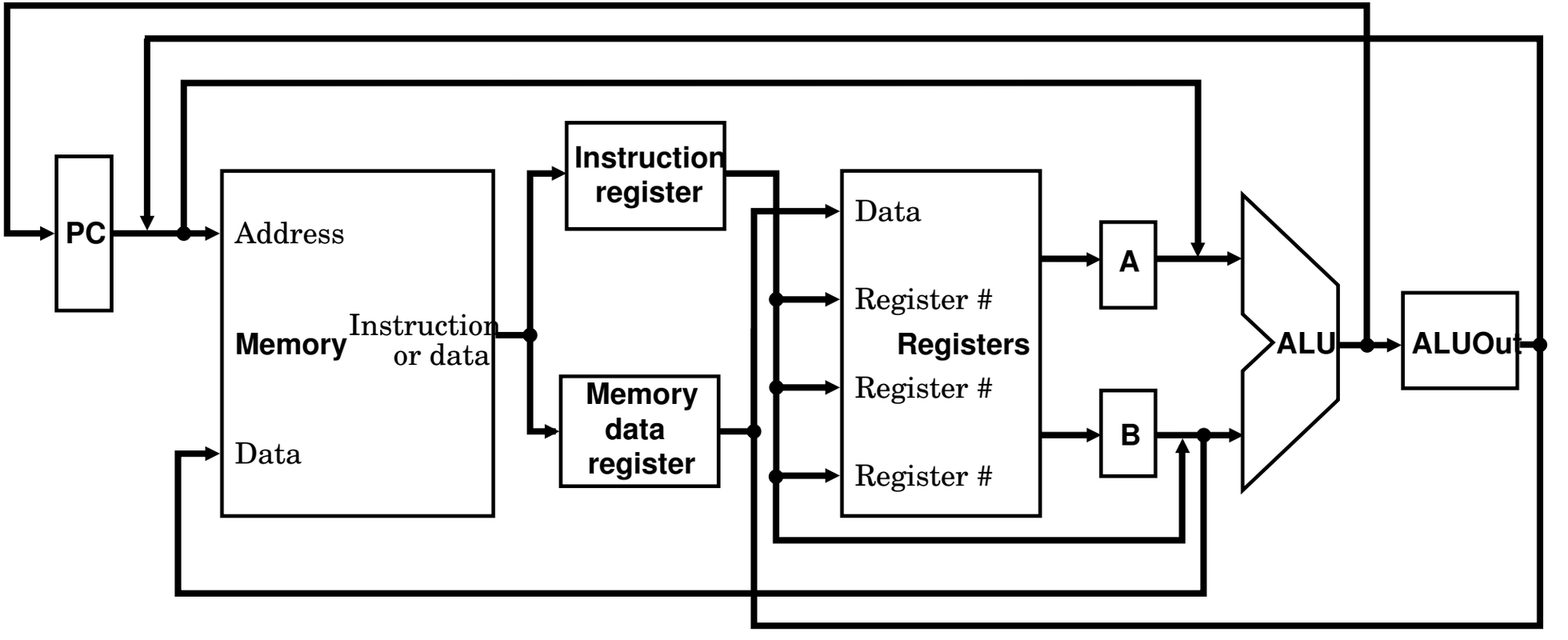
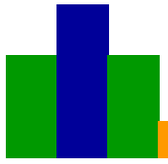


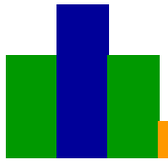
Figura conţine elementele de bază ale unui procesor pentru *cicluri multiple*. De remarcat noii regiştrii auxiliari IR (Instruction Register), MDR (Memory Data Register), A, B, ALUOut, pentru reţinerea datelor de la un ciclu la altul, în procesarea aceleaşi instrucţiuni.



Registri auxiliari

Registri auxiliari:

- Datele necesare *instrucţiunilor care urmează* se reţin în elemente de memorie specifice: IM, DM, ori RF.
- Datele necesare *în acceaşi instrucţiune* în cicluri diferite, necesită regiştri speciali de memorare.
- Folosim următorii regiştri auxiliari:
  - *IR* (*Instruction Register*) - pentru instrucţiunea citită şi *MDR* (*Memory Data Register*) - pentru data citită;
  - *A* şi *B* - pentru valorile citite din regiştri RF;
  - *ALUOut* (*ALU Output*) - pentru rezultatul ALU.



..Registri auxiliari

Registri auxiliari (cont.)

De notat că:

- Registrul *IR* reţine instrucţiunea *până la finalul execuţiei*.
- In afară de IR, *ceilalţi regiştri* reţin datele doar *de la un ciclu la următorul*.

Reutilizarea componentelor de la un ciclu la altul necesită:

- *multiplexoare noi* (or expandate);
- un *control extins*.

..Implementare cu cicluri multiple

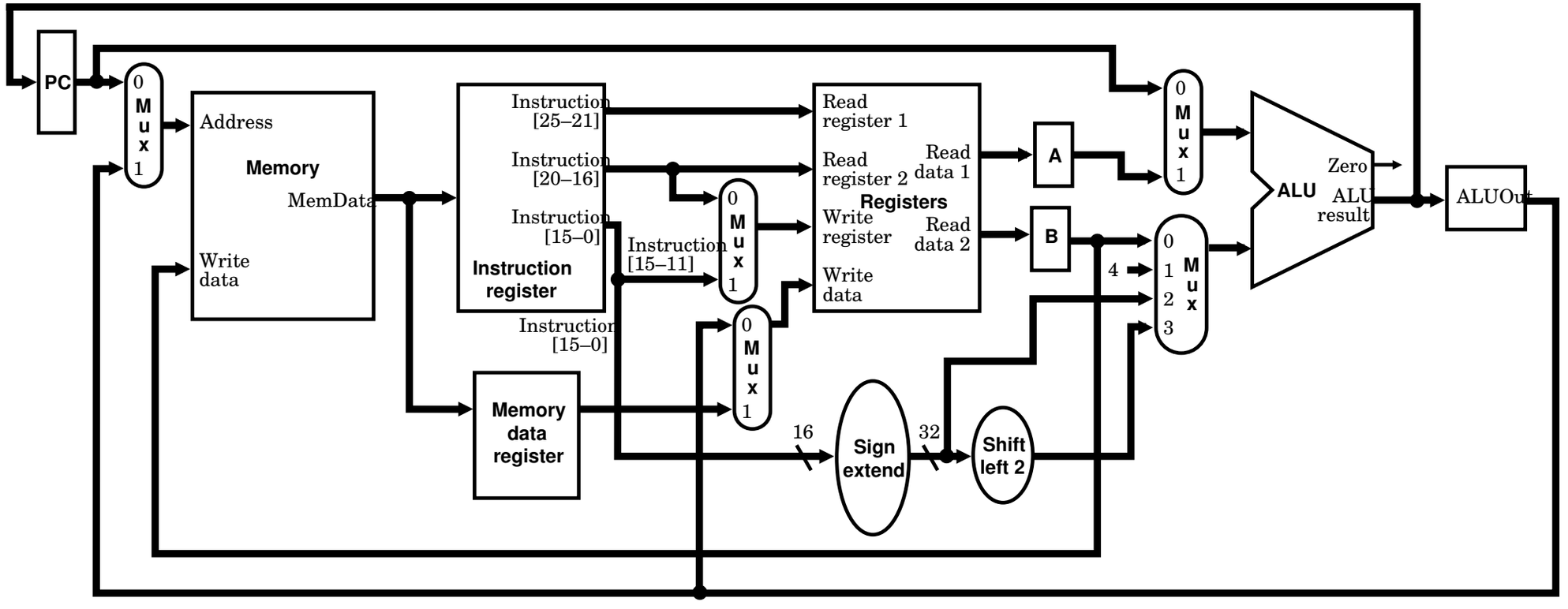
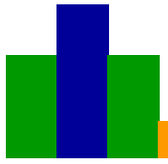


Figura conţine schema de procesor cu cicluri multiple şi *multiplexoarele noi* necesare pentru procesarea instrucţiunilor MIPS (fără branch şi jump).



Multiplexoare suplimentare

Multiplexoare suplimentare:

- Pentru înglobarea sumatoarelor în ALU, adăugăm
  - un *multiplexor la primul argument* (care alege între PC şi registrul A);
  - un *multiplexor cu 4 selecţii la argumentul al doilea* (noile intrări sunt pentru constanta "4" şi ieşirea `Shift 2` de după `Sign Extend`).

  Economie: un multiplexor este mult mai ieftin decât un ALU!
- Pentru unificarea memoriilor adăugăm
  - un *multiplexor la memorie* (care alege între acces date ori instrucţiuni)

  Economie similară, unificând memoriile (deşi capacitatea trebuie sporită).

..Implementare cu cicluri multiple

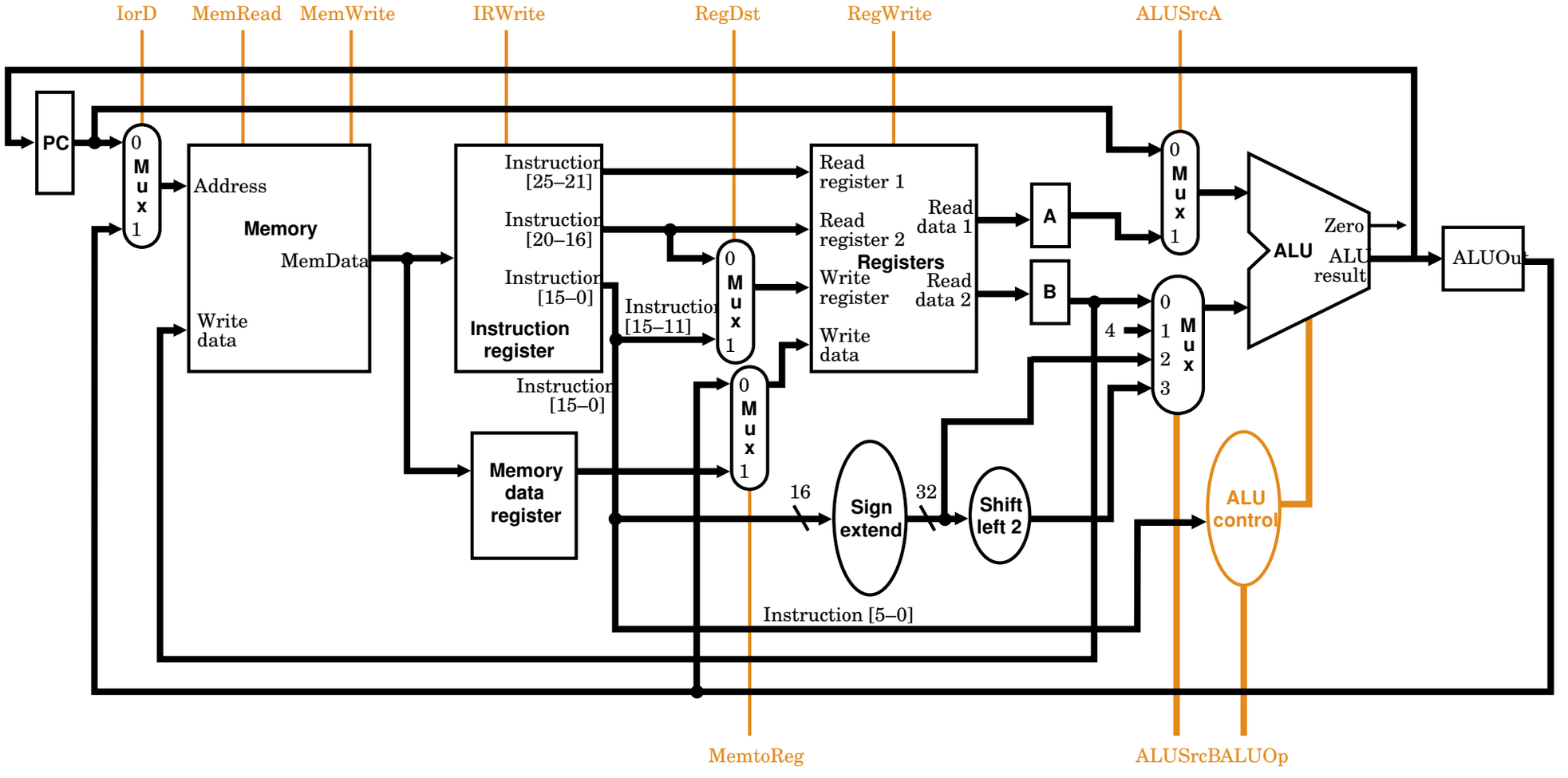
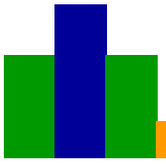


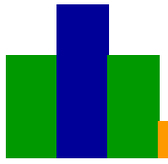
Figura conţine schema de procesor cu cicluri multiple şi *semnalele de control* necesare pentru procesarea instrucţiunilor MIPS (fără branch şi jump).



Semnale de control suplimentare

Semnale de control suplimentare:

- `IorD` (1b) - alege dacă se citeşte din memorie o instrucţiune ori o dată;
- `IRWrite` (1b) - indică când se scrie noua instrucţiune în registru;
- `ALUSrcA` (1b) - alege de unde vine primul argument ALU;
- `ALUSrcB` (2b) - alege de unde vine al doilea argument ALU;



Includem: branch + jump

Includem: branch + jump:

Folosim *un multiplexor cu 3 poziţii* care indică de unde vine sursa pentru *noul PC*, anume:

- *din ALU*, dacă este instrucţiunea următoare $\texttt{PC} + 4$;
- din *registrul* `ALUOut`, dacă ramificaţia indicată de branch este acceptată şi adresa vine ca rezultat al adresării indirecte (registru de bază + deplasare);
- direct *din instrucţiune* (26b, shift-aţi cu 2 poziţii şi suprapuşi peste poziţiile 27-0 din PC), dacă instrucţiunea următoare este indicată de un jump.

..Implementare cu cicluri multiple

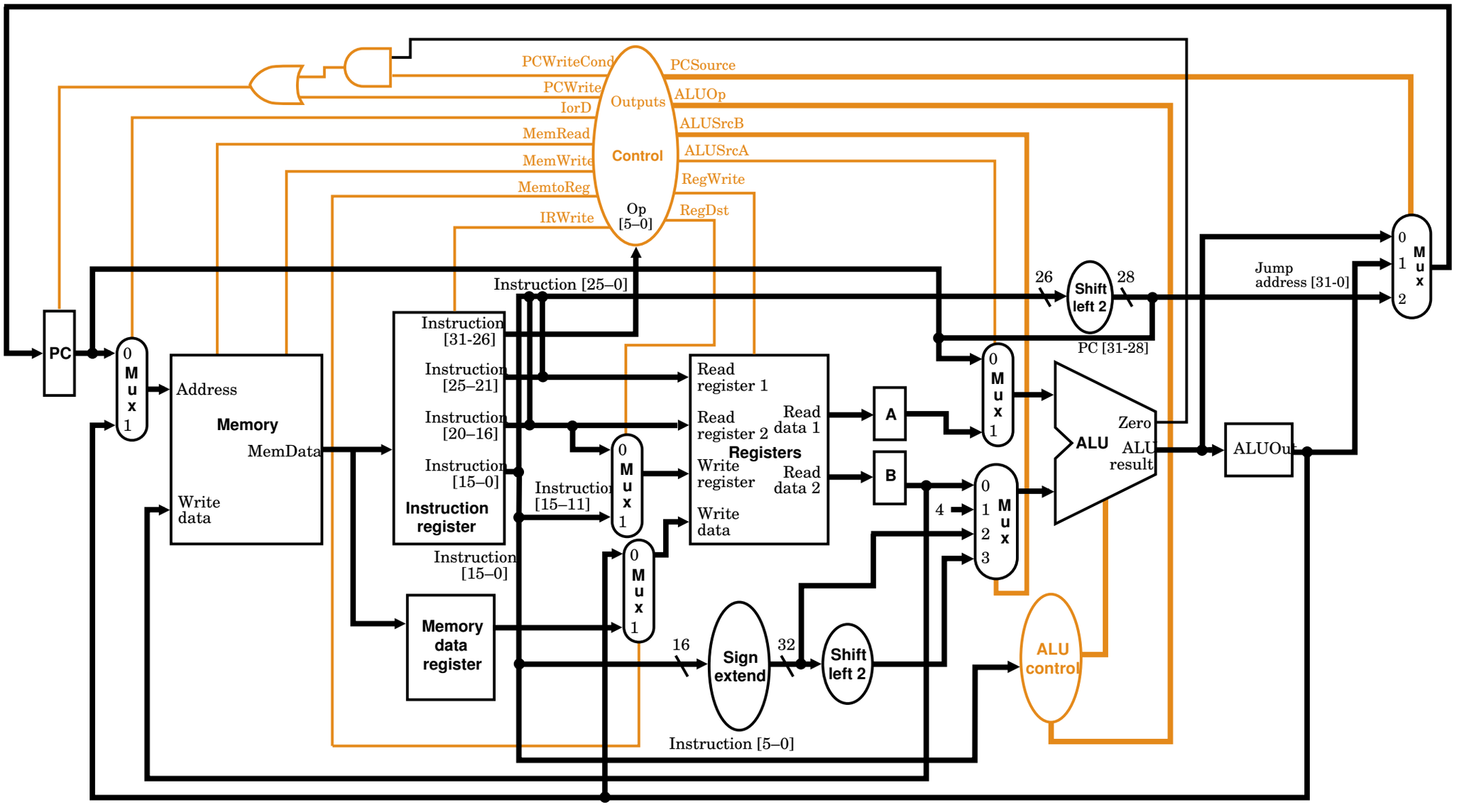
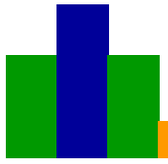


Figura conţine *schema completă de procesor cu cicluri multiple* (incluzând branch şi jump şi unităţile de control aferete.



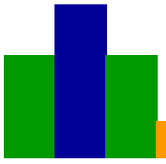
Semnale de control suplimentare

Semnale de control suplimentare: In fine, ultimile semnale de control adăugate sunt pentru a *scrie noul PC*:

- Folosim fie
  - `PCWrite` (1b), care indică când se scrie noul PC în memorie în cazul necondiţionat (PC + 4 or jump);

  fie
  - `PCWriteCond` (1b), care indică când se scrie noul PC în memorie în cazul condiţionat, folosind, în plus, ieşirea de test `zero` din ALU.
- In figură este inserat un circuit simplu care combină semnalele de mai sus în mod adecvat, anume cel specificat de

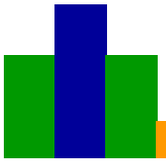
  $\phi =$ `(zero AND PCWriteCond) OR PCWrite.`



Semnalele de control (final)

Semnalele de control pe 1b:

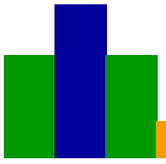
Semnal	Efect la desetare	Efect la setare
`RegDst`	Desetat: registrul destinatie pentru scriere vine din campul `rt`	Setat: registrul destinatie pentru scriere vine din campul `rd`
`RegWrite`	Desetat: nimic	Setat: se scrie data in registrul destinatie
`ALUSrcA`	Desetat: primul argument este PC	Setat: primul argument vine din registrul A
`MemRead`	Desetat: nimic	Setat: continutul de memorie de la adresa de la intrare este pus la iesire
`MemWrite`	Desetat: nimic	Setat: data de intrare este scrisa in memorie la adresa de la intrare
`MemtoReg`	Desetat: valoarea returnata pentru scriere in registru vine din ALU	Setat: valoarea returnata pentru scriere in registru vine din MDR / memorie



..Semnalele de control (final)

Semnalele de control pe 1b (cont.)

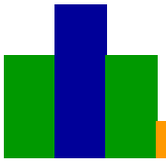
Semnal	Efect la desetare	Efect la setare
`IorD`	Desetat: adresa de accesat memoria vine din PC	Setat: adresa de accesat memoria vine din ALUOut
`IRWrite`	Desetat: nimic	Setat: iesirea din memoria se scrie in IR
`PCWrite`	Desetat: nimic	Setat: se scrie PC (sursa fiind controlata de `PCSource`)
`PCWriteCond`	Desetat: nimic	Setat: se scrie PC daca si iesirea `zero` din ALU este setata



Semnalele de control (final)

Semnalele de control pe 2b:

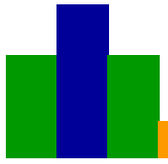
Semnal	Valoare	Efect
`ALUOp`	00	ALU efectueaza o adunare
	01	ALU efectueaza o scadere
	11	ALU efectueaza o operatie indicata de campul `field` din instructiune
`ALUSrcB`	00	al 2-lea argument ALU vine din registrul B
	01	al 2-lea argument ALU este 4
	01	al 2-lea argument ALU este extensia cu semn a ultimilor 16b din IR
	11	al 2-lea argument ALU este extensia cu semn a ultimilor 16b din IR, shiftata cu 2b
`PCSource`	00	iesirea ALU (PC+4) este trimisa la PC pentru scriere
	01	ALUOut (adresa branch-ului) este trimisa la PC pentru scriere
	10	adresa jump-ului (IR[25-0], shiftat cu 2b, si completat cu PC+4[31-28]) este trimisa la PC pentru scriere



Impartirea executiei instructiunii in cicluri

Generalitati:

- Imparţitrea execuţiei intrucţiunilor în cicluri trebuie făcută *echilibrat* (sarcini egale în fiecare ciclu), pentru a scurta perioada ceasului.
- Unele date trebuiesc reţinute de la un ciclu la altul în *regiştri auxiliari*; cu tehnologia noastră de ceas (i.e., schimbare pe front de undă), valorile noi apar abia la noul ciclu de ceas.
- Unele operaţii se execută în *paralel* în acelaşi ciclu (e.g., simultan se incrementează PC-ul şi se accesează memoria).
- De notat că regiştri din RF (Register File) necesită *un ciclu în plus* faţă de regiştrii suplimentari ori PC. (Motiv: RF e mai complex, cu semnale de control complicate, accesul său fiind mai greoi; cu această tehnică, perioada de ceas se scurtează.)



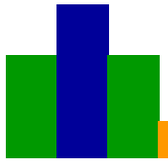
..Impartirea executiei instructiunii in cicluri

Pasul 1. Extragerea instructiunii:

- *Sarcini:* extragem instrucţiunea din memorie şi calculăm adresa instrucţiunii următoare

```
IR = Memory[PC];          (1)
PC = PC + 4;              (2)
```

- *Implementare:*
  - (1) selectăm `IorD = 0` şi setăm semnalele `MemRead`, `IRWrite`;
  - (2) selectăm `AlUSrcA = 0`, `AlUSrcB = 01`, `AlUOp = 00` (adunare), `PCSource = 00` şi setăm `PCWrite`.



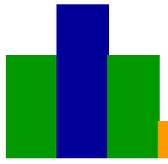
..Impartirea executiei instructiunii in cicluri

Pasul 2. Decodarea instructiunii şi extragerea regiştrilor:

- *Sarcini:* Facem fie paşi comuni la toate instrucţiunile (citim `rs`), fi parţial comuni, dar care nu dăunează celor diferite (citim `rt` şi calculăm adresa de salt la branch);

```
A = Reg[IR[25-21]];                                  (1)
B = Reg[IR[20-16]];                                  (2)
ALUOut = PC + (sign-extend (IR[15-0]) << 2);         (3)
```

- *Implementare:*
  - (1) şi (2) nimic special [se citesc regiştrii `rs,rt` şi se rescriu în `A,B` *în fiecare ciclu*];
  - (3) selectăm `AlUSrcA = 0`, `AlUSrcB = 11`, `AlUOp = 00` (adunare); adresa de salt se reţine în `ALUOut`.



..Impartirea executiei instructiunii in cicluri

Pasul 3. Operaţie ALU, calcul adresă, ori completare branch: In acest ciclu operaţia este definită de instrucţiune:

- *Referinţa la memorie:*

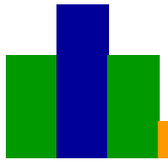
```
ALUOut = A + sign-extend (IR[15-0]);
```

Implementare: selectăm `AlUSrcA = 1`, `AlUSrcB = 10`, `AlUOp = 00` (adunare);

- *Operaţie atitmetică ori logică (tip R):*

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

Implementare: selectăm `AlUSrcA = 1`, `AlUSrcB = 00`, `AlUOp = 10` (operaţia ALU rezultă din câmpul `field` via `ALUControl`);



..Impartirea executiei instructiunii in cicluri

Pasul 3. (cont.):

- *Branch:*

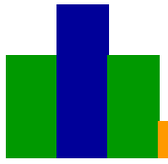
```
if (A == B) PC = ALUOut;
```

Implementare: Folosim adresa de salt din `ALUOut` (calculată în Pasul 2) şi testul `zero` din ALU:
selectăm `AlUSrcA = 1`, `AlUSrcB = 00`, `AlUOp = 01` (scădere), `PCSource = 01` şi setăm `PCWriteCond`
[Notă: rescriem peste adresa PC + 4 scrisă în pasul 1.];

- *Jump:*

```
PC = PC[31-28] || (IR[25-0)<<2);
```

Implementare: selectăm `PCSource = 10` şi setăm `PCWrite`.



..Impartirea executiei instructiunii in cicluri

Pasul 4. Acces memorie ori completare instrucţiune tip R:

- *Referinţa la memorie:*

```
Memory[AluOut] = B;          (1)
```

ori

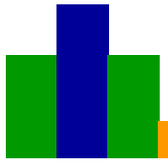
```
MDR = Memory[ALUOut];        (2)
```

Implementare: (1) selectăm `IorD = 1` şi setăm `MemWrite` (`B` a fost folosit şi anterior; este acelaşi `B`, căci `IR` nu se schimbă); (2) selectăm `IorD = 1` şi setăm `MemRead` (`MRD` se scrie în fiecare ciclu);

- *Operaţie aritmetică ori logică (tip R):*

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

Implementare: selectăm `RegDst = 1`, `MemtoReg = 0` şi setăm `RegWrite`.



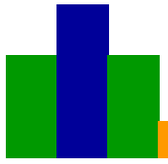
..Impartirea executiei instructiunii in cicluri

Pasul 5. Completare acces memorie: Doar instrucţiunea load a rămas necompletată.

- *Referinţa la memorie (load):*

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

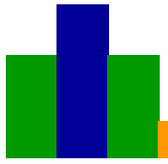
Implementare: selectăm `MemtoReg = 1`, `RegDst = 0` şi setăm `RegWrite`.



..Impartirea executiei instructiunii in cicluri

Tabel general:

Pas	Instr. tip R	Referinta mem.	Branch	Jump
1	IR = Memory[PC]; PC = PC + 4;			
2	A = Reg[IR[25-21]]; B = Reg[IR[20-16]]; ALUOut = PC + (sign-extend (IR[15-0]) << 2);			
3	ALUOut = A op B;	ALUOut = A + sign-extend (IR[15-0]);	if (A == B) PC = ALUOut;	PC = PC[31-28] || (IR[25-0)<<2);
4	REG[IR[15-11]] = ALUOut;	Memory[AluOut] = B; (store) MDR = Memory[ALUOut]; (load)		
5		Reg[IR[20-16]] = MDR; (load)		



Definirea controlului

Definirea controlului:

- Se folosesc *porţi*, *ROM*-uri, ori *PLA*-uri pentru a sintetiza unitatea de control.
- Folosim *maşini cu stări finite* pentru reprezentarea grafică a unităţii de control.
  [Alternativ, se poate folosi *microprogramarea*.]
- Tehnică: Determinăm funcţia care dă *starea următoare* şi funcţia de *ieşire* care, apoi, se implementează cu un circuit cu memorie.
- Convenţii: (1) Semnalele de control ca nu sunt *explicit setate*, se consideră *nesetate*.
  (2) La multiplexoare, *setarea la 0* poate fi subînţeleasă, deci nu necesită porţi.

..Definirea controlului

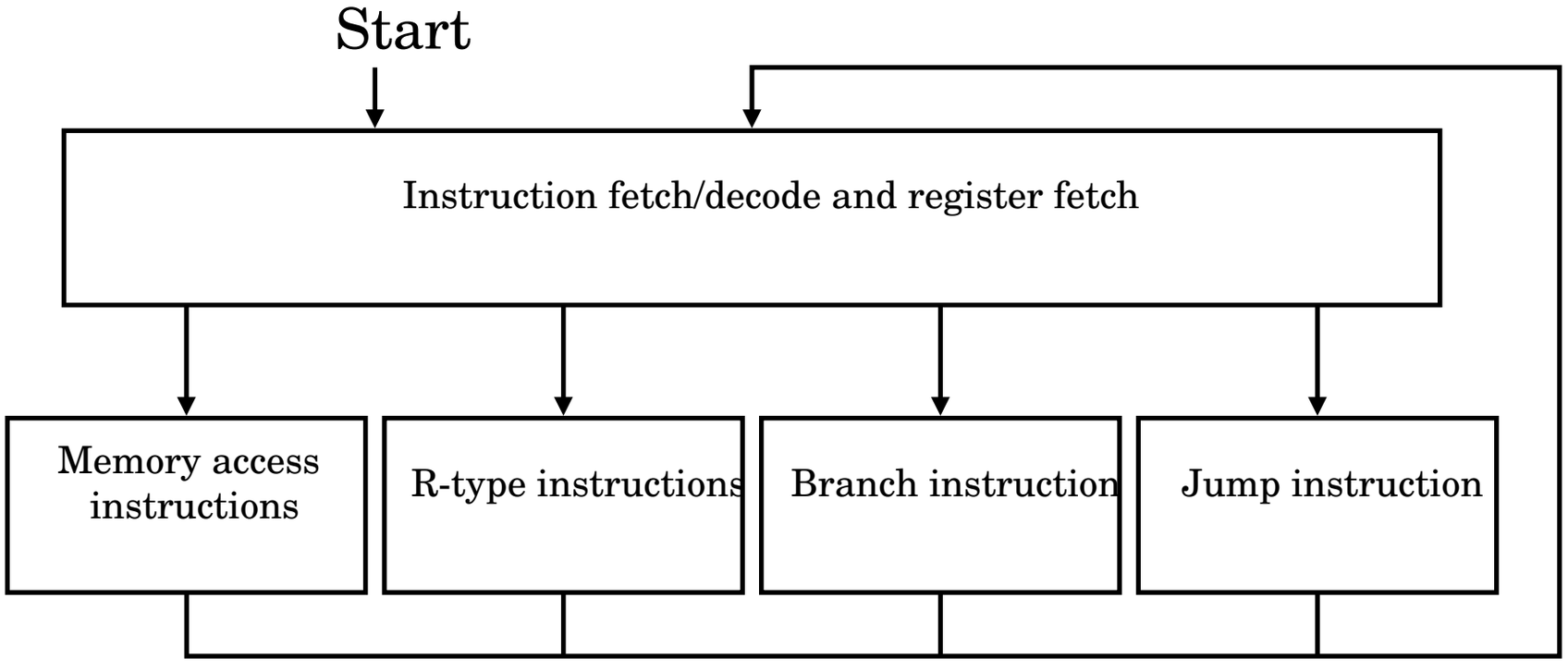


Figura conţine *schema generală a unităţii de control*. Vom prezenta separat o componentă de control pentru fiecare tip de instrucţiune, pe care le asamblăm ca în figură.

..Definirea controlului

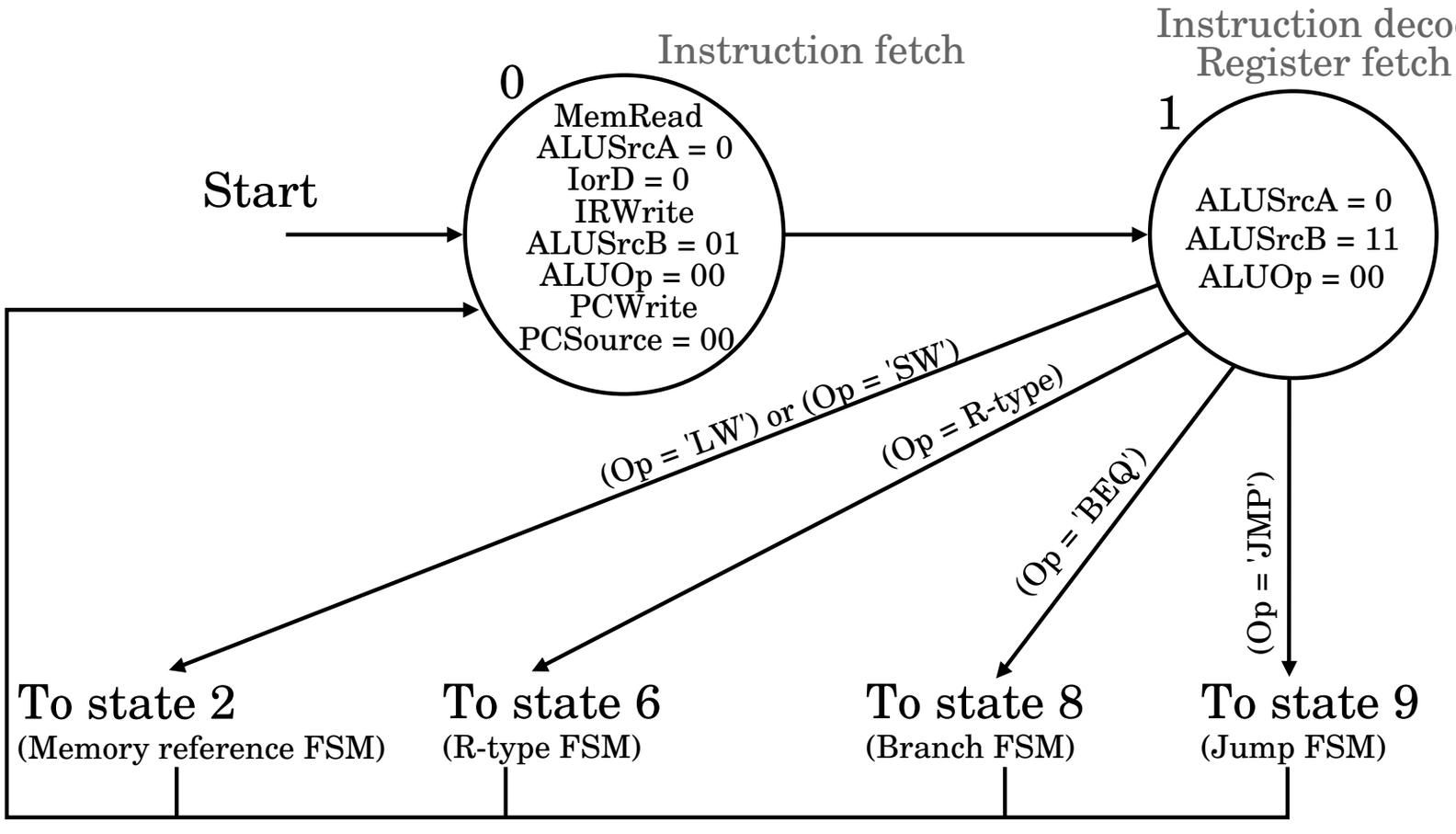


Figura conţine componentele comune pentru controlul fazelor 1 & 2: *extragere a instrucţiunii* şi *decodificare*.

..Definirea controlului

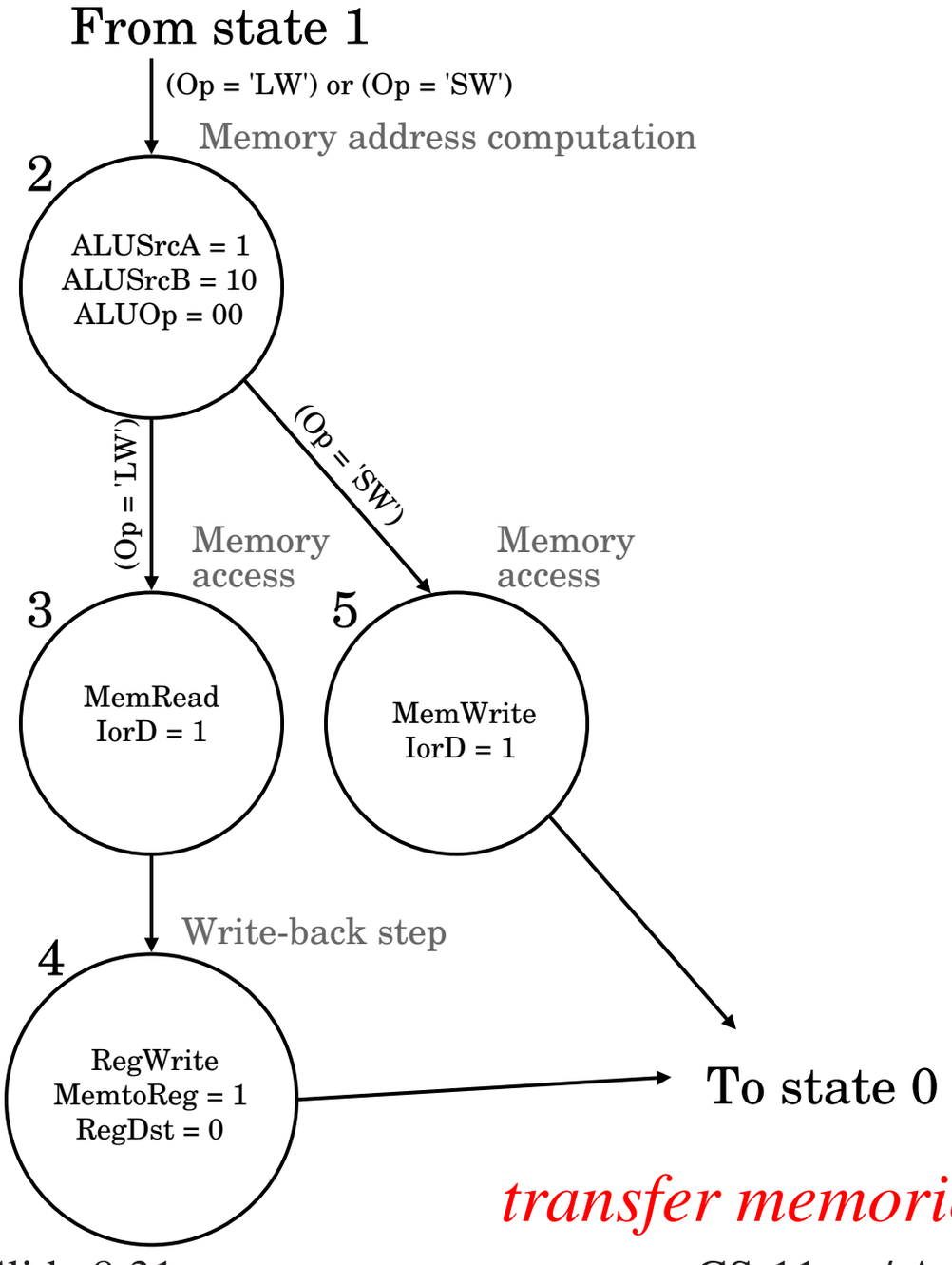


Figura conţine controlul pentru procesare separată a instrucţiunilor (fazele 3,4,5): *transfer memorie*, operaţii de tip R, branch şi jump.

..Definirea controlului

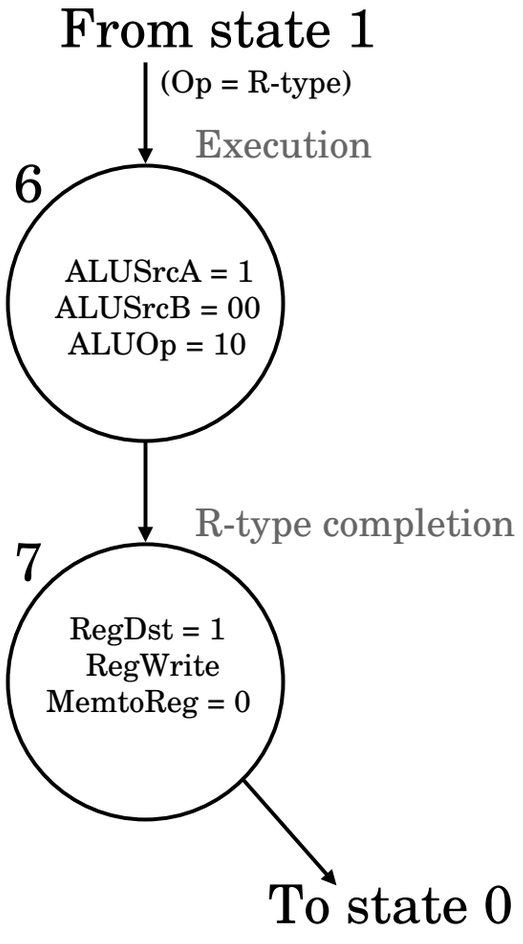


Figura conţine controlul pentru procesare separată a instrucţiunilor (fazele 3,4,5): transfer memorie, *operaţii de tip R*, branch şi jump.

..Definirea controlului

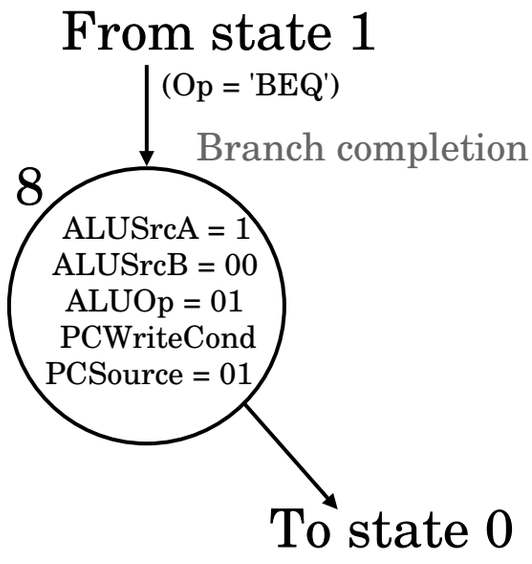


Figura
conţine controlul
pentru procesare separată
a instrucţiunilor (fazele 3,4,5):
transfer memorie, operaţii de tip R, *branch* şi jump.

..Definirea controlului

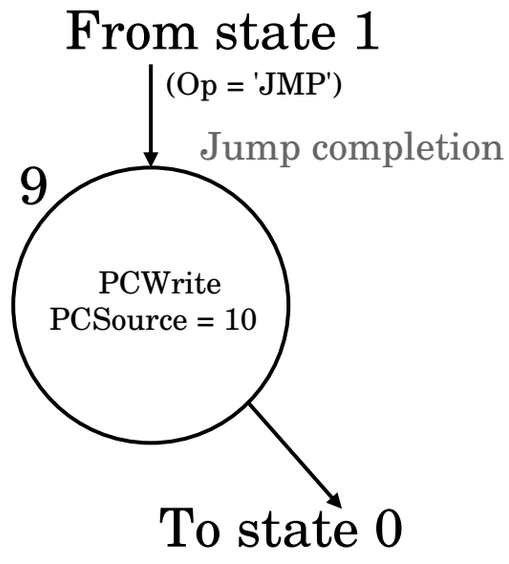
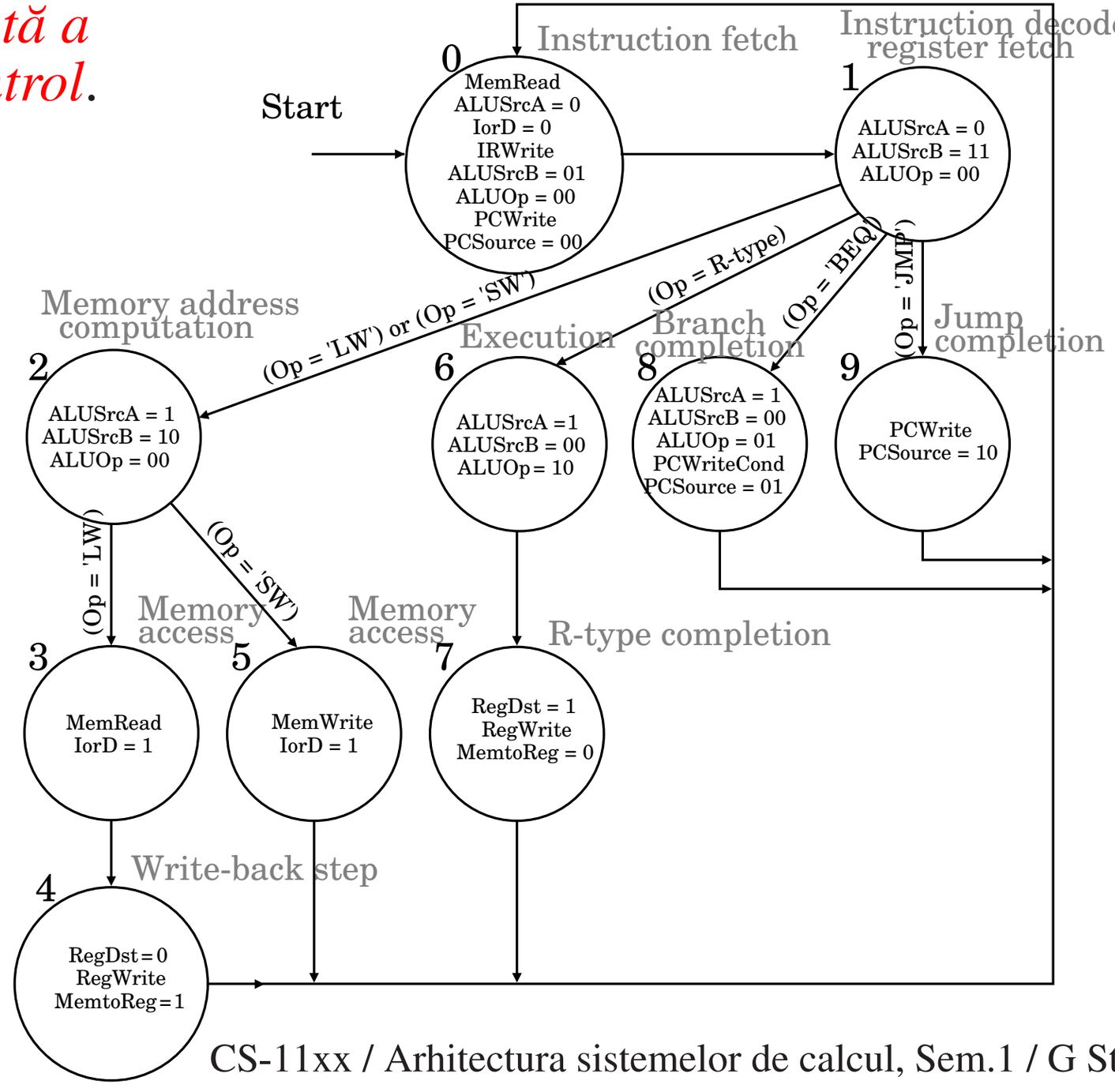


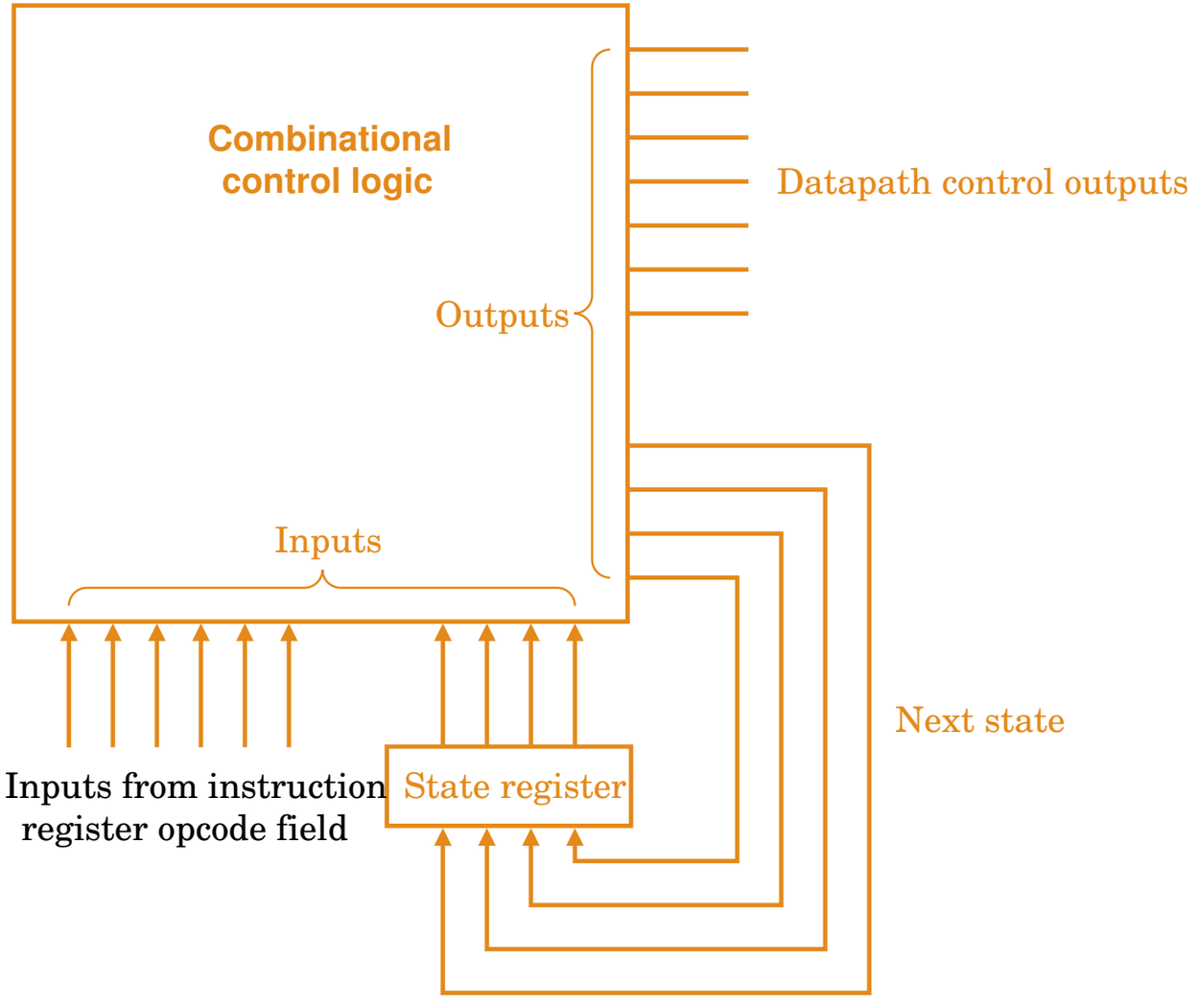
Figura
conţine controlul
pentru procesare separată
a instrucţiunilor (fazele 3,4,5):
transfer memorie, operaţii de tip R, branch şi *jump*.

..Definirea controlului

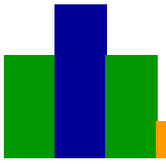
Figura detaliată a unităţii de control.



..Definirea controlului



Implementarea unităţii de control folosind o *unitate combinaţională* şi *elemente de memorie (regiştri)* pentru starea curentă.



Performanta implementarii cu cicluri multiple

Performanta, cicluri multiple: Implementarea cu un cicluri multiple nu este mult mai eficientă:

- Pe o mixtură de 43% ALU, 23% load, 13% store, 19% branch, 2% jump (tipică pentru compilatorul `gcc`) reducerea este de la

  $$\texttt{CPU} = 5 \text{ (implementare cu un singur ciclu)}$$

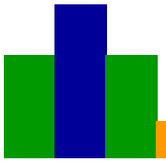
  la

  $$\begin{aligned}\texttt{CPU} &= 4 \times 0.43 + 5 \times 0.23 + 4 \times 0.13 + 3 \times 0.19 + 3 \times 0.02 \\ &= 4.02\end{aligned}$$

- Performanţa obţinută este:

  $$\texttt{Perf mciclu}/\texttt{Perf ciclu} = 5/4.02 = 1.24$$

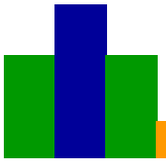
Câştigul enorm va veni de la combinarea acestei implementări cu tehnica de *pipeline* (prezentată în lecţia următoare).



Procesorul: Calea de date si controlul

Cuprins:

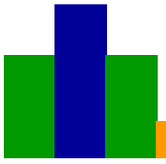
- Generalitati
- Calea de date
- O prima implementare
- Implementare cu cicluri multiple
- *Microprogramare*
- Exceptii
- Concluzii, diverse, etc.



Microprogramare

Generalitati:

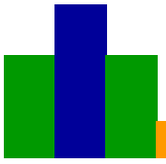
- Maşinile cu stări finite sunt adecvate pentru *cazuri simple*, precum subsetul nostru MIPS cu 9 instrucţiuni.
- Trebuie căutate soluţii alternative pentru *cazuri complexe*. Exemplu: MIPS (complet) are peste 100 instrucţiuni, implementarea lor variind între 1 şi 20 de cicluri pe instrucţiune.
- Unităţile de controlul rezultate pot ajunge la *mii de stări*, cu sute de secvenţe diferite (exemplu: setul Intel 80x86).
- Pentru simplificare, folosim *microprogramarea* care indică cum se setează şi desetează semnalele de control; instrucţiunile folosite se numesc *microinstrucţiuni*.



Microinstructiuni

Microinstructiuni:

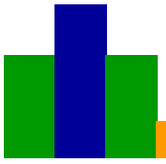
- O microinstrucţiune foloseşte *în paralel* mai multe câmpuri disjuncte.
- Microinstrucţiunile se pun în ROM ori PLA, deci au o adresă.
- Alegerea instrucţiunii următoare se face prin:
  - *Incrementarea* adresei curente (dacă valorea câmpul Sequencing este `Seq`);
  - Salt la adresa pentru procesarea unei *noi instrucţiuni MIPS* (dacă valorea câmpul Sequencing este `Fetch`);
  - Salt la adresa dată de o *unitate locala de control* - distributie (dispatch) folosind o tabelă de adrese (dacă valorea câmpul Sequencing este `Dispatch i`);
- O *microinstructiune* este un simplu tuplu de 8 valori.



Microinstructiuni

Campurile microinstructiunilor:

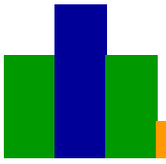
Numele campului	Functia campului
ALU control	Specifica operatia ALU executata in ciclul curent; rezultatul se scrie in ALUOut.
SRC1	Specifica sursa primului argument ALU.
SRC2	Specifica sursa celui de-al doilea argument ALU.
Register control	Specifica o operatie de scriere ori de citire in RF (Register File); la scriere, specifica si sursa datei scrise.
Memory	Specifica o operatie de scriere ori de citire in memorie; la scriere, specifica si sursa datei scrise; la citire, specifica si destinatia datei citite.
PCWrite control	Specifica scrierea PC-ului.
Sequnecing	Specifica cum se alege noua microinstructiune.



..Microinstructiuni

Campurile microinstructiunilor si valori:

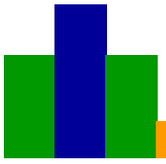
Nume camp	Valori camp	Functia campului cu valorile specificate
Label	Orice string	Etichetele sunt folosite pentru saltul in microprogram. Etichetele care se termina cu 1 ori 2 sunt pentru distributie cu tabel de salt, indexat de codul operatiei. Celelalte sunt pentru salt direct. Etichetele nu genereaza semnale de cotrol direct, ci indirect via instructiunea Sequencing.
ALU Control	`Add`	Seteaza ALU sa adune.
	`Subt`	Seteaza ALU sa scada (bun si pentru comparatii).
	`Func code`	Utilizeaza codul `field` din instructiune pentru ALU control.
SRC1	`PC`	Primul argument ALU este PC.
	`A`	Primul argument ALU este din registrul A.
SRC2	`B`	Al 2-lea argument ALU este din registrul B.
	`4`	Al 2-lea argument ALU este 4.
	`Extend`	Al 2-lea argument ALU este din `Sign-Extend`.
	`Extshft`	Al 2-lea argument ALU este din `Shift left 2`.



..Microinstructiuni

Campurile microinstructiunilor si valori (cont.)

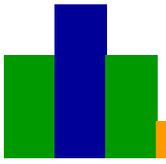
Nume camp	Valori camp	Functia campului cu valorile specificate
Register control	`Read`	Citeste registri specificati de campurile `rs,rt` din `IR` si scrie in registri `A,B`.
	`Write ALU`	Scrie in registrul specificat de campul `rd` din `IR` data din registrul `ALUOut`.
	`Write MDR`	Scrie in registrul specificat de campul `rt` din `IR` data din registrul `MDR`.
Memory	`Read PC`	Citeste memoria cu adresa din PC si scrie in registrul `IR` (si in `MDR`, dar nefolosit).
	`Read ALU`	Citeste memoria cu adresa din `ALUOut` si scrie in registrul `MDR`.
	`Write ALU`	Scrie in memorie la adresa din `ALUOut` continutul din registrul `B`.



..Microinstructiuni

Campurile microinstructiunilor si valori (cont.)

Nume camp	Valori camp	Functia campului cu valorile specificate
PCWrite control	`ALU`	Scrie iesirea ALU in PC.
	`ALUOut-cond`	Daca iesirea `zero` din ALU este 1, scrie continutul registrului `ALUOut` in PC.
	`Jump address`	Scrie in PC adresa jump din instructiune.
Sequnecing	`Seq`	Se continua cu urmatoarea instructiune.
	`Fetch`	Se trece la prima microinstructiune (pentru a procesa o noua instructiune).
	`Dispatch i`	Executia urmatoare este definita de ROM-ul specificat de `i` (1 ori 2).



Microprograme

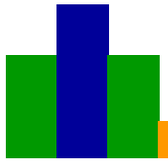
Microprograme:

Paşii 1 & 2 de *extractie instrucţiune* şi *decodare* revin la:

Label	ALU	SRC1	SRC2	Register	Memory	PCWrite	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	ExtShft	Read			Despatch 1

Paşii 3-4 ori 3-5 de la *accesarea memoriei* revin la:

Label	ALU	SRC1	SRC2	Register	Memory	PCWrite	Sequencing
Mem1	Add	A	Extend				Dispatch 2
LW2					Read ALU		Seq
				Write MDR			Fetch
SW2					Write ALU		Fetch



..Microprograme

Microprograme (cont.)

Paşii 3-4 la *instructiunea de format R* revin la

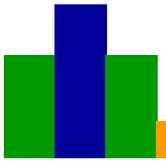
Label	ALU	SRC1	SRC2	Register	Memory	PCWrite	Sequencing
Rformat1	Func code	A	B				Seq
				Write ALU			Fetch

Pasul 3 de la *instrucţiunea conditională* revine la:

Label	ALU	SRC1	SRC2	Register	Memory	PCWrite	Sequencing
BEQ1	Subt	A	B			ALUOut-cond	Fetch

Pasul 3 de la *instrucţiunea de salt* revine la:

Label	ALU	SRC1	SRC2	Register	Memory	PCWrite	Sequencing
JUMP1						Jump address	Fetch



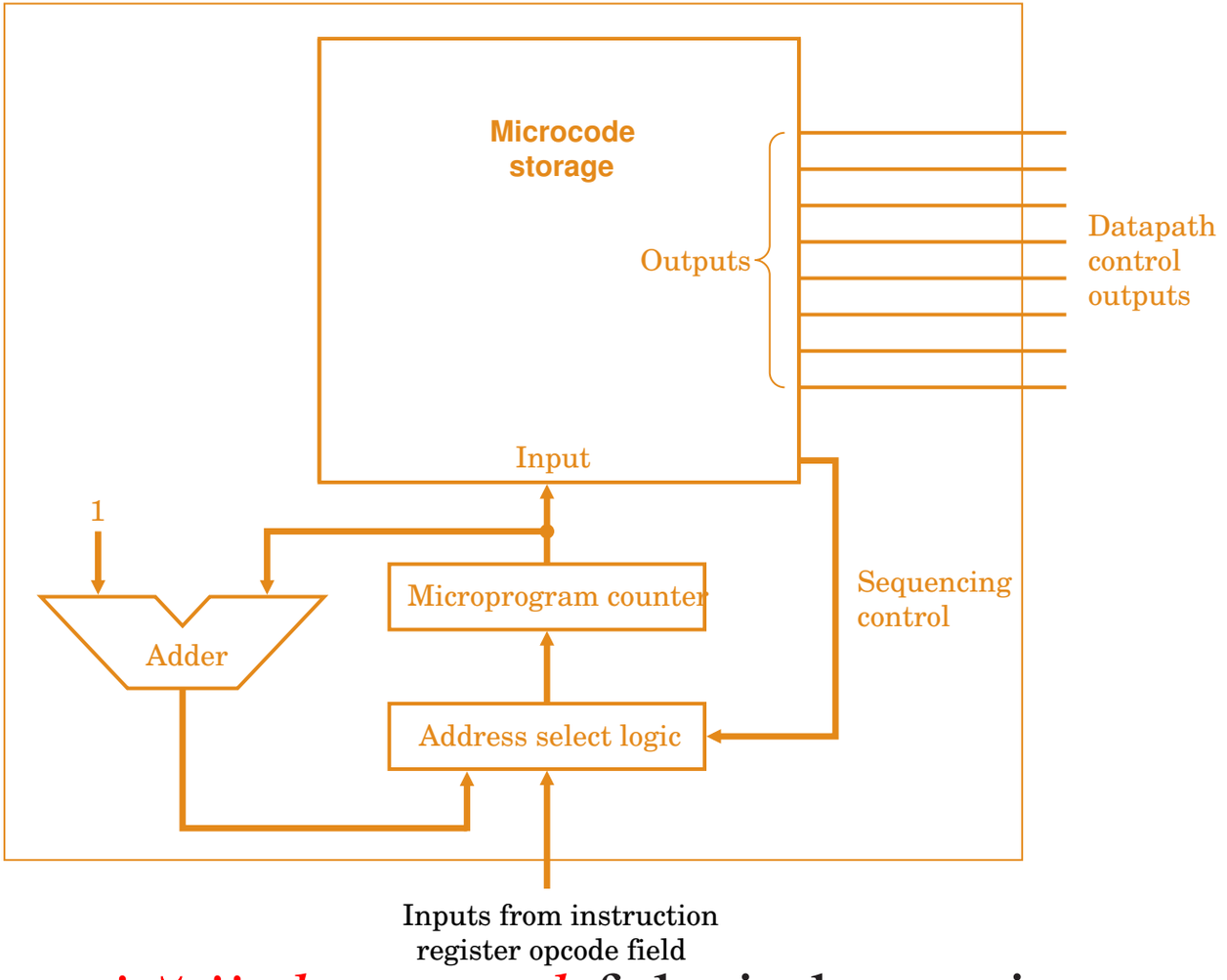
..Microprograme

Microprograme (cont.)

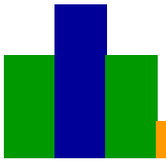
Programul complet este

Label	ALU	SRC1	SRC2	Register	Memory	PCWrite	Sequencing
Fetch	Add	PC	4		Read PC	ALU	Seq
	Add	PC	ExtShft	Read			Despatch 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

Microprogramare: Implementare



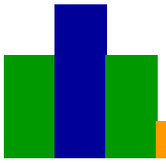
Implementarea unităţii de control folosind un microprogram; *microcodul* este implementat cu un ROM ori PLA; *contorul* microprogram selecteaza noua instructiune; el este definit de un circuit de *selectie a adresei*.



Procesorul: Calea de date si controlul

Cuprins:

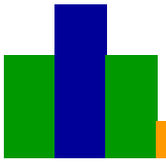
- Generalitati
- Calea de date
- O prima implementare
- Implementare cu cicluri multiple
- Microprogramare
- *Exceptii*
- Concluzii, diverse, etc.



Exceptii

Exceptii:

- Partea cea mai dificilă din procesor este *controlul* - dificil de făcut *corect şi rapid*.
- In particular, este dificilă implementarea *excepţilor* şi a *întreruperilor*, care afectează cursul normal de procesare (diferă de branch şi jump):
  - excepţiile sunt evenimente neaşteptate *din procesor*, e.g., overflow, etc.;
  - întreruperile sunt evenimente similare *din afara procesorului*, e.g., comunicări I/O, etc.
- Uneori se foloseşte acelaşi termen de “excepţie” pentru ambele categorii de evenimente.



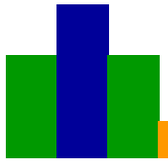
..Exceptii

Exemple:

Tip de eveniment	Unde apare	Clasificare MIPS
Cerere de la un dispozitiv I/O	Extern	Intrerupere
Invocarea sistemului de operare dintr-un program utilizator	Intern	Exceptie
Depasire aritmetica (overflow)	Intern	Exceptie
Folosire de instructiune nedefinita	Intern	Exceptie
Eroare de functionare a hardware-ului	Oricare	Exceptie ori Intrerupere

Actiuni:

- Acţiunea de bază la excepţie este de a salva adresa instrucţiunii care a generat-o într-un registru special *EPC* (*Exception Program Counter*).
- După procesarea excepţiei, sistemul de operare poate folosi EPC spre a continua execuţia.



..Exceptii

Identificarea cauzei:

- MIPS foloseste un registru de stare *Cause Register* din care se deduce cauza.
- Sistemul de operare poate avea *întreruperi vectorizate*, anume multiple puncte de intrare care depind de cauza întreruperii şi produc o procesare specifică limitată.
- Dacă nu este ca mai sus, se intră printr-un *unic punct* şi sistemul de operare află cauza din registrul de stare înainte de a procesa întreruperea.
- Pnetru controlul noilor regiştri se folosesc noi semnale de control `EPCWrite` şi `CauseWrite`.

..Implementare cu cicluri multiple

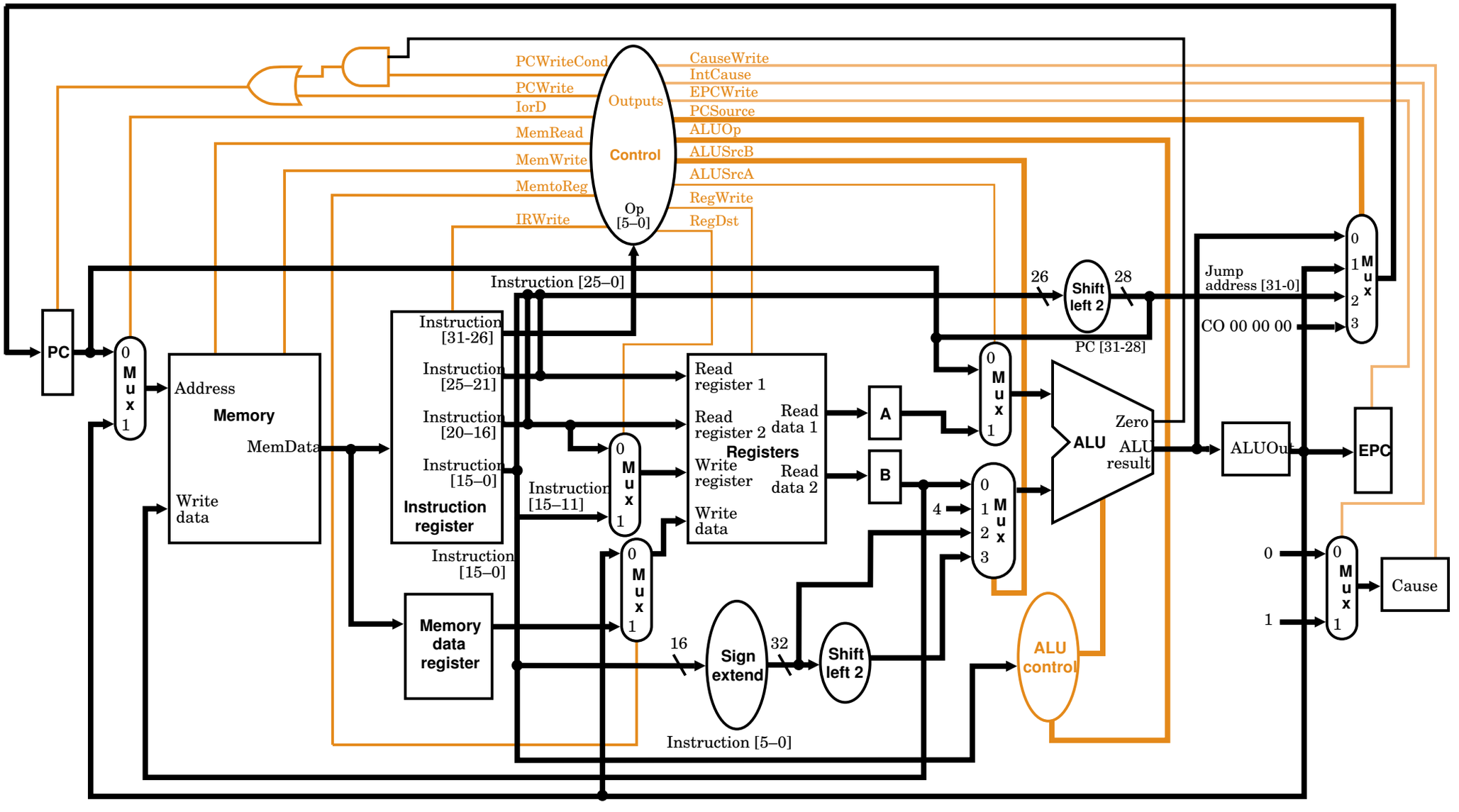


Figura conţine *schema de procesor cu cicluri multiple*, extinsă cu procesarea a *2 tipuri de excepţii* (*operaţii nedefinite* şi *overflow*).

..Definirea controlului

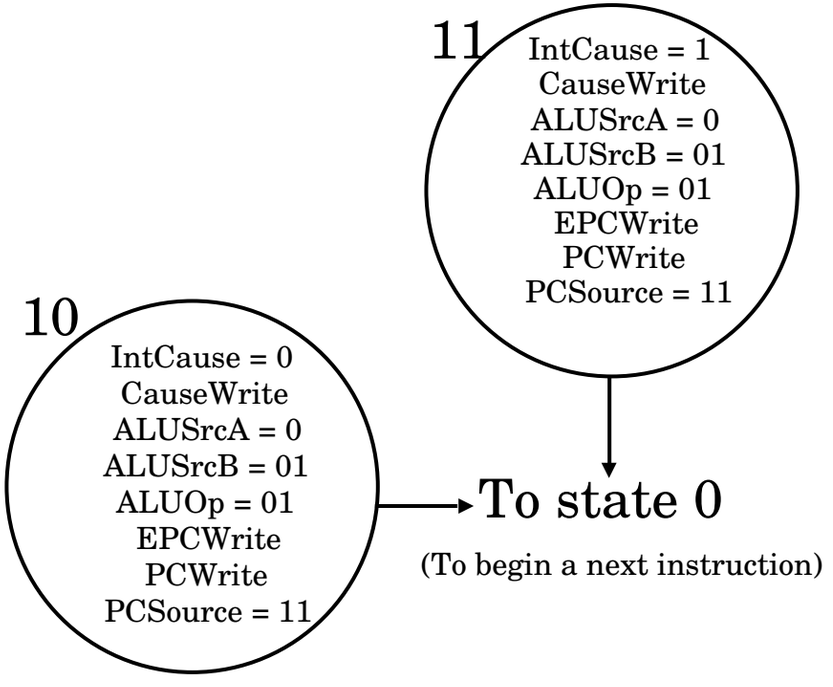
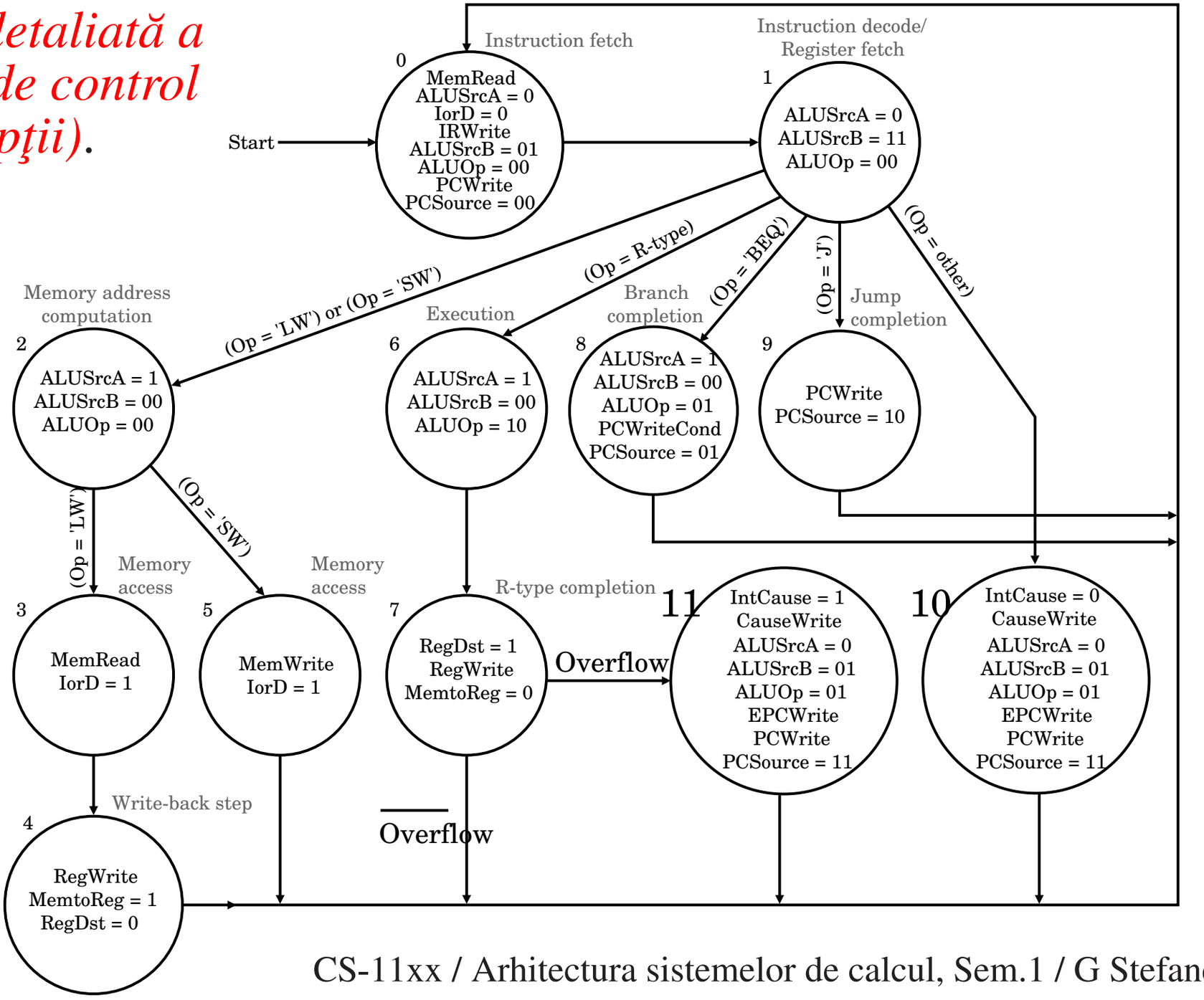
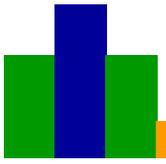


Figura conţine controlul pentru procesare separată a excepţiilor considerate: *depăşire (overflow)* şi *instrucţiune nedefinită*.

..Definirea controlului

Figura detaliată a unităţii de control (cu excepţii).





Procesorul: Calea de date si controlul

Cuprins:

- Generalitati
- Calea de date
- O prima implementare
- Implementare cu cicluri multiple
- Microprogramare
- Exceptii
- *Concluzii, diverse, etc.*