Latvijas Universitāte
Datorikas fakultāte

Daudztaktu procesors (nobeigums).

Kurss "Ievads digitālajā projektēšanā"

Lekcija 10.12.2010

Autors: **Artis Mednis**

Rīga 2010

Kontroles modulis

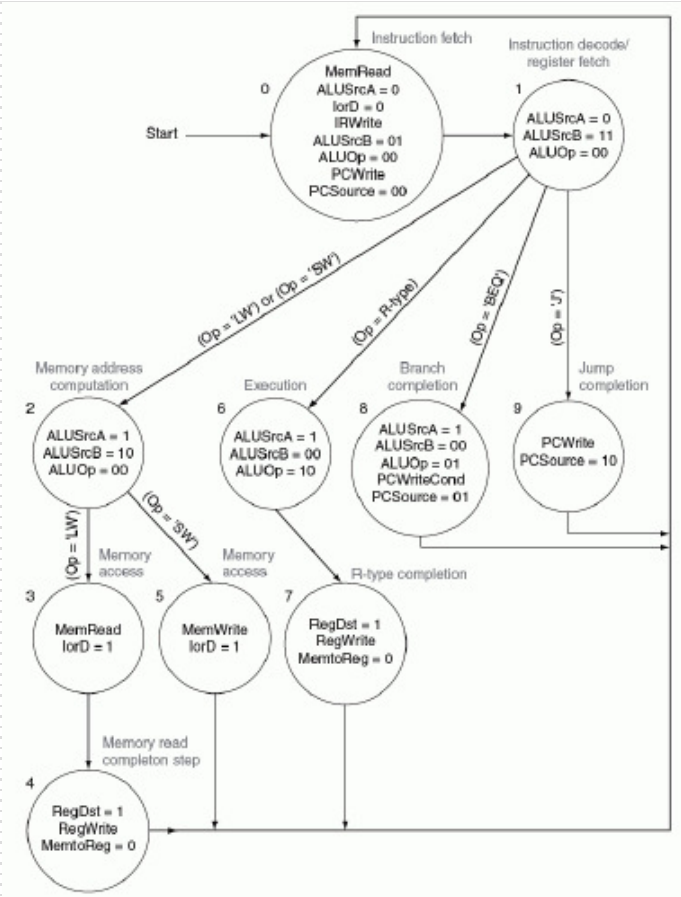
- Vientakts procesors
  - Kombinacionālā loģika, kas balstās uz instrukcijas teksta saturu
- Daudztaktu procesors
  - Instrukcijas tiek izpildītas vairākos (3..5) soļos
  - Jāņem vērā dati, kas tiek iegūti, izpildot tekošo soli, kā arī nākošais veicamais solis
- Divi atšķirīgi risinājumi
  - Galīgie automāti (stāvokļi un pārejas starp tiem)
  - Mikroprogrammēšana (*mikroinstrukcijas*, kas paredzētas izpildei uz vienkāršas *mikromašīnas*)

Galīgie automāti

- Sastāvdaļas
  - Stāvokļi
  - Pārejas starp stāvokļiem
- Nākošā stāvokļa funkcija – aktuālais stāvoklis un ieejas parametri nosaka nākošo stāvokli
- Izejas funkcija – aktuālais stāvoklis un (iespējams) ieejas parametri nosaka izejas parametrus
- Kontrole tiek realizēta, noteiktos stāvokļos uzstādot kā 1 noteiktus kontroles signālus (pārējā laikā šiem signāliem piešķir vērtību 0)

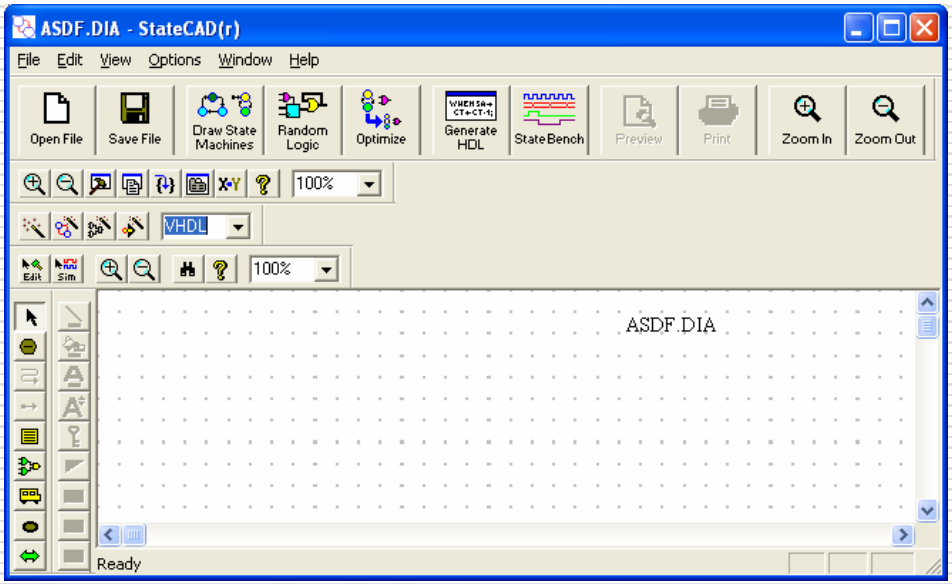
Kontroles realizācija, izmantojot galīgos automātus

- Pirmie divi stāvokļi (0 & 1) kopīgi visām instrukcijām
- Pārējie stāvokļi (izpildes secība) atšķirīgi, atkarībā no instrukcijas tipa
- Katrs stāvoklis atbilst vienai taktij



Galīgie automāti un *Xilinx ISE*

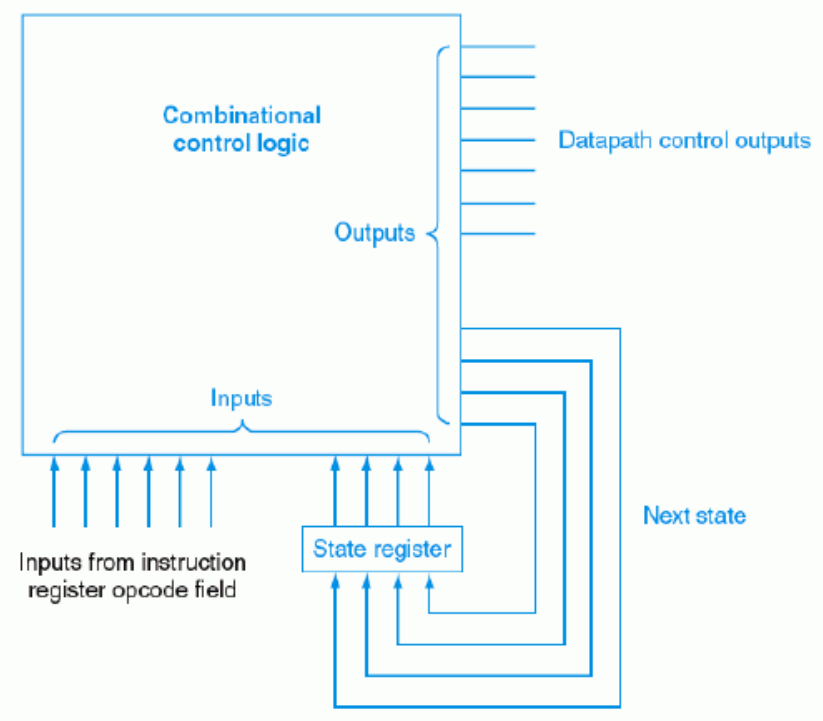
- Labā ziņa ☺
  - Ir rīks *StateCAD*, kurā var zīmēt stāvokļu pārejas diagrammas un no tām sintezēt *HDL* kodu



- Sliktās ziņas ☹
  - Rīks pieejams tikai *Xilinx ISE Windows* versijā
  - Kopš versijas 10.1 vairs nav pieejams

Fiziskā implementācija

- Reģistrs aktuālā stāvokļa glabāšanai
- Kombinacionālā loģika vadības signālu un nākošā stāvokļa uzstādīšanai

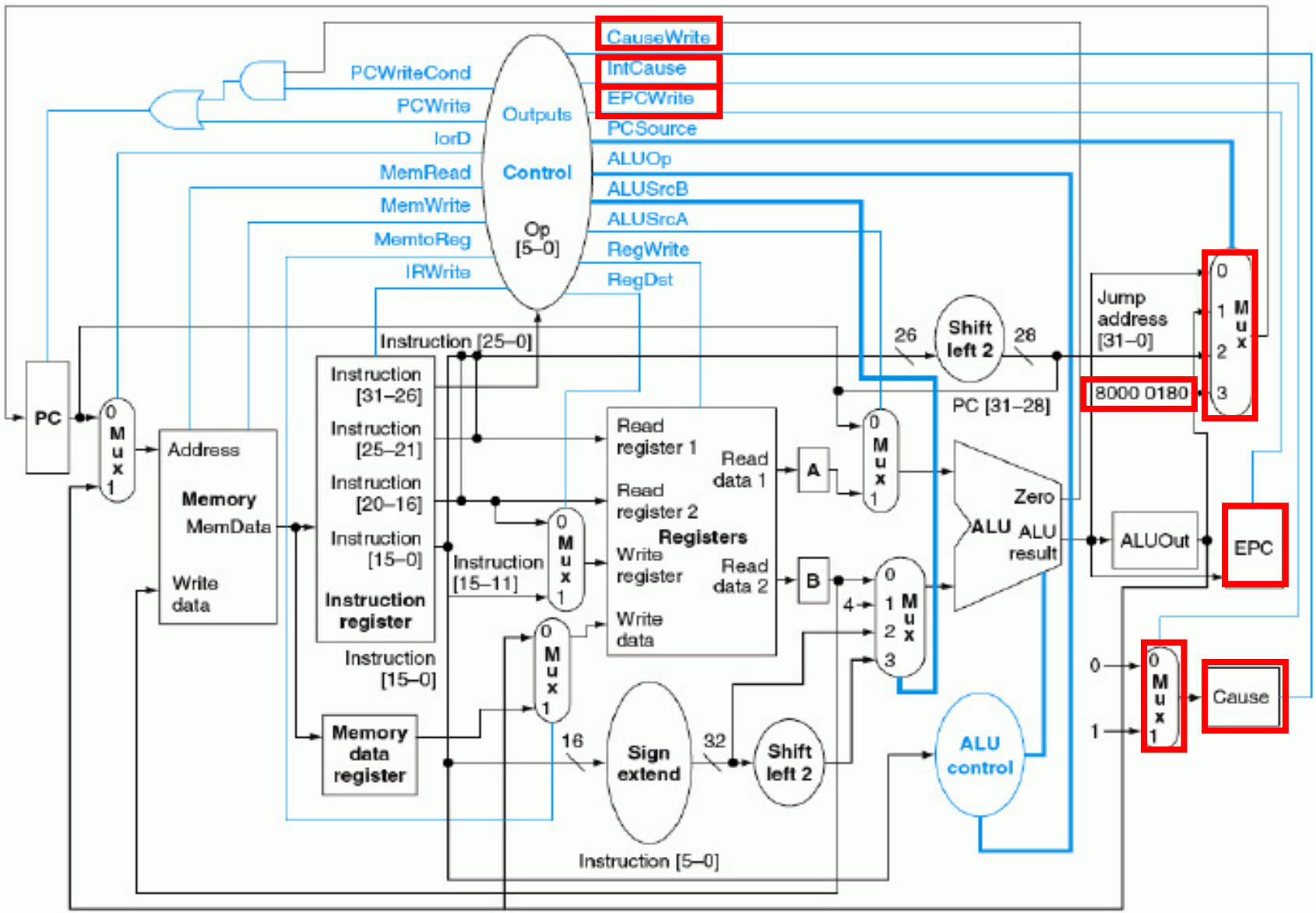


Izņēmumi un pārtraukumi

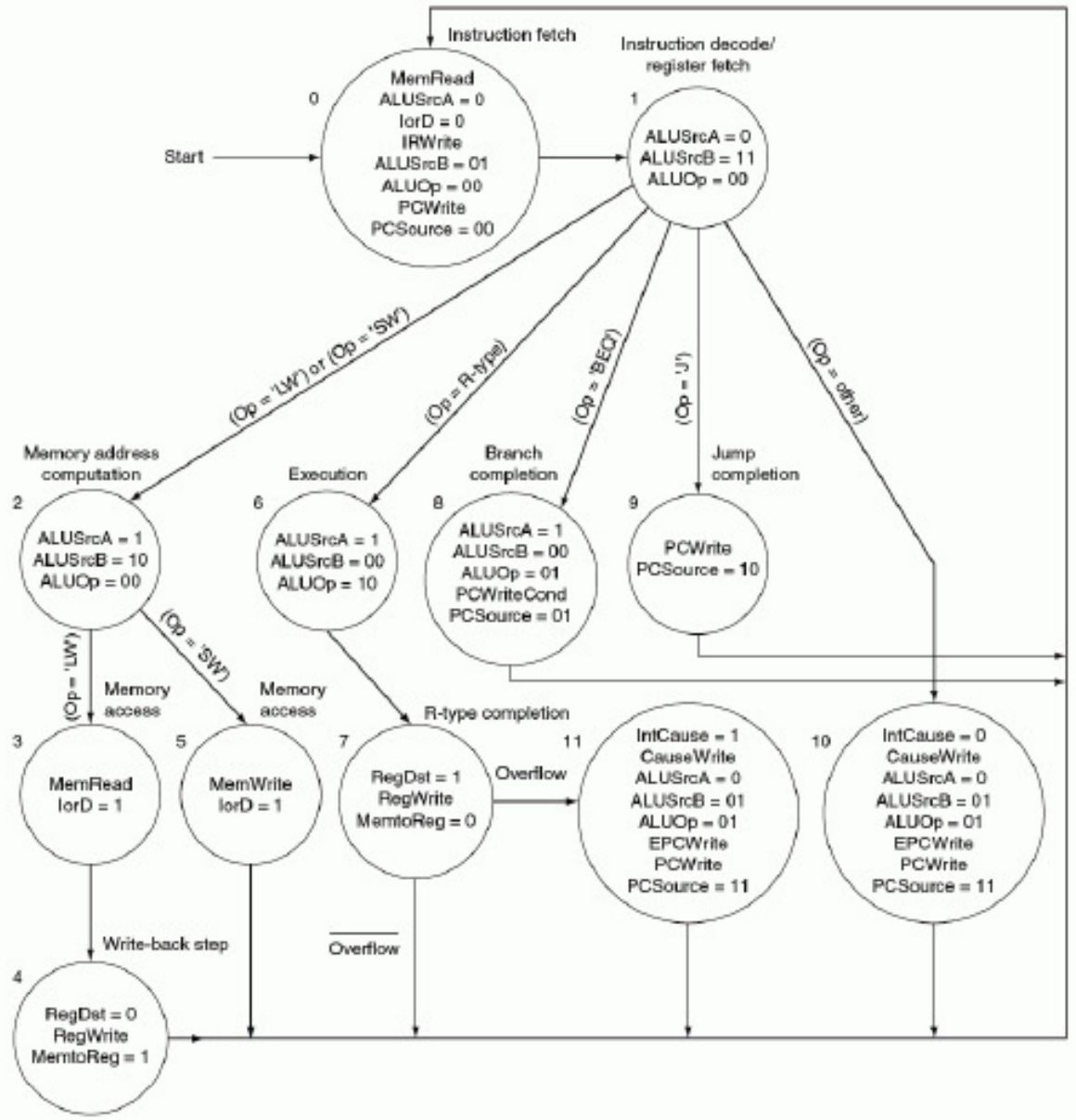
- Izņēmums (*exception*)
  - negaidīts notikums procesora *iekšpusē*, kura dēļ jāmaina programmas izpilde
  - piemērs – aritmētiskā pārpilde
- Pārtraukums (*interrupt*)
  - negaidīts notikums procesora *ārpusē*, kura dēļ jāmaina programmas izpilde
  - piemērs – kādas *I/O* ierīces *vēršanās* pie procesora
- Atkarībā no arhitektūras un *ticības*, izņēmums un pārtraukums var būt/var nebūt viens un tas pats

Problēmu detektēšana un apstrāde

- Dažas tipveida problēmas
  - Nepazīstama instrukcija
  - Aritmētiska pārpilde
- Rīcības algoritms
  - Saglabāt izpildāmās instrukcijas adresi speciālā reģistrā *EPC* (*exception program counter*)
  - Nodot vadību operētājsistēmai, kura var apturēt programmas darbību, kā arī (iespējams) to atsākt, izmantojot *EPC* saglabāto adresi
- Operētājsistēmai ir *jāzin*, kāds ir problēmas iemesls
  - Statusa reģistrs – dažādiem iemesliem dažāds saturs
  - Vektora pārtraukums - dažādiem iemesliem dažādas pārejas adreses


CauseWrite
IntCause
EPCWrite
PCWriteCond
PCWrite
IorD
MemRead
MemWrite
MemtoReg
IRWrite
Outputs
Control
Op
[5–0]
PCSource
ALUOp
ALUSrcB
ALUSrcA
RegWrite
RegDst
Instruction [25–0]
26
Shift
left 2
28
Jump
address
[31–0]
PC [31–28]
8000 0180
PC
Address
Memory
MemData
Write
data
Instruction
[31–26]
Instruction
[25–21]
Instruction
[20–16]
Instruction
[15–0]
Instruction
register
Instruction
[15–0]
Instruction
[15–11]
Read
register 1
Read
register 2
Registers
Write
register
Write
data
Read
data 1
Read
data 2
A
B
4
Zero
ALU
ALU
result
ALUOut
EPC
Cause
Memory
data
register
16
Sign
extend
32
Shift
left 2
ALU
control
Instruction [5–0]
M
u
x

- Nepazīstama instrukcija tiek atpazīta, izejot no stāvokļa 1
- Aritmētiska pārpilde tiek atpazīta, izejot no stāvokļa 7



- Pārtraukums 10 minūtes

Mikroprogrammēšana I

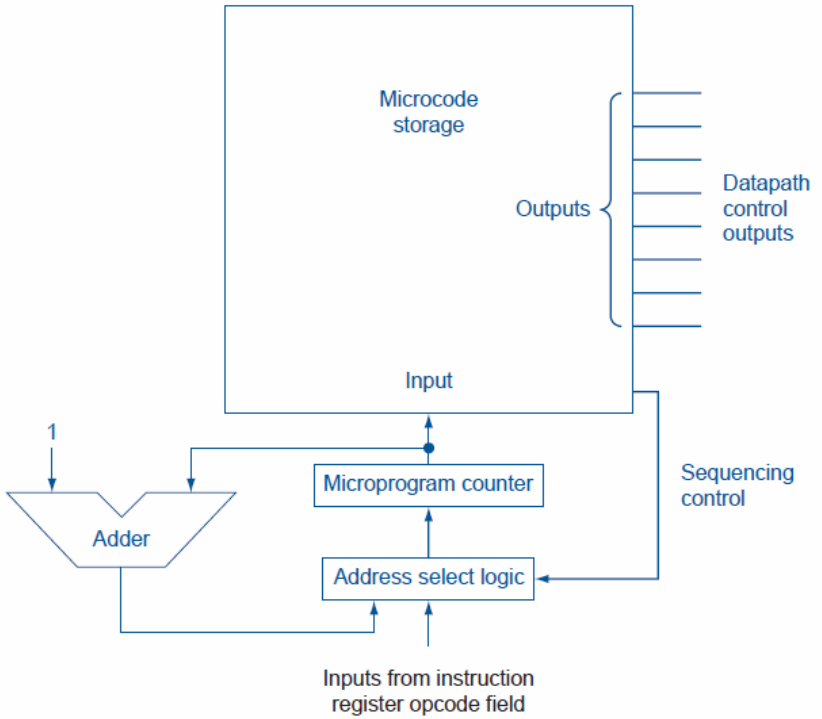
- Kas tad ir *slikti* galīgajiem automātiem? (diskusija)
  - Pārdesmit instrukcijas un pārdesmit stāvokļi lieliski ietilpst vienā lapā
  - Nopietnas instrukciju kopas satur vairākus simtus instrukciju
  - Katra no tām var tikt izpildīta no viena līdz pat vairāk kā 20 takts cikliem
  - Līdz ar to jārēķinās ar tūkstošiem iespējamo stāvokļu un simtiem iespējamo izpildes ceļu
  - Tas vairs nav projektējams grafiskā veidā ☹
  - Ko lai iesāk?

Mikroprogrammēšana II

- Varētu aizņemties dažas idejas no klasiskās programmēšanas
  - Katram stāvoklim varētu atbilst mikroinstrukcija, kas uzstāda noteiktu kontroles signālu kombināciju
  - Katrai zarošanai varētu atbilst mikroinstrukcija, kas kontrolē ieejas parametrus un veic atbilstošo pāreju
  - Lai ekonomētu uz koda apjomu, būtu labi, ja varētu izmantot apakšprogrammas (koda atkalizmantošana)

Mikroprogrammēšana III

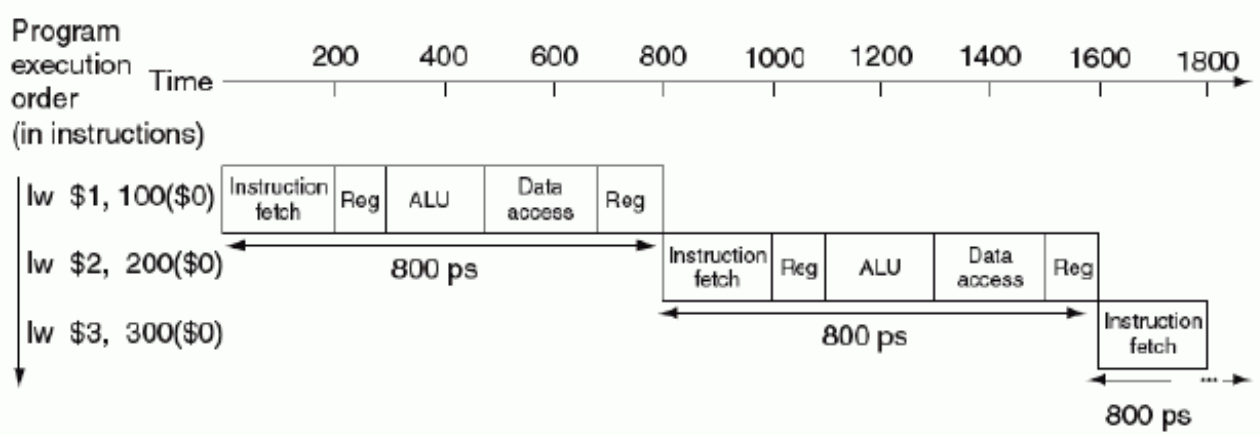
- Fiziskā realizācija
  - Pastāvīgā atmiņa (*ROM*)
  - Programmējamie loģiskie masīvi (*PLA*)
- Nākošo instrukciju izvēlas vienā no 3 veidiem
  - Nākošā pēc kārtas (pēc noklusējuma)
  - Kārtējās MIPS instrukcijas izpilde (stāvoklis 0)
  - Atbilstoši ieejas signāliem (attiecīgās adreses arī glabājas kādā tabulā *ROM* vai *PLA*)

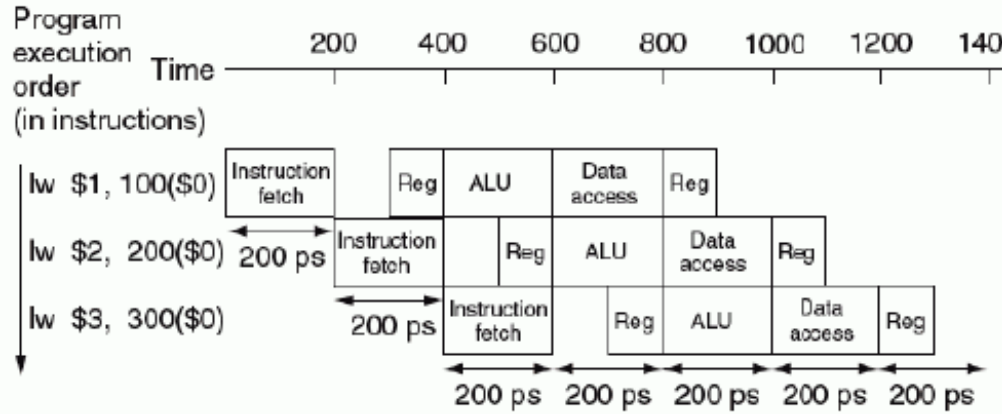


Ātrāk, vēl ātrāk...

- Kā vēl varētu paātrināt procesora darbību?
- Pirmais solis bija instrukciju vientakts izpildes nomaiņa uz daudztaktu izpildi
- Kas notiks, ja mēs tiem *DataPath* komponentiem, kas ir tikuši galā ar savu darbu, dosim nākošo darba uzdevumu, neskatoties uz to, ka instrukcijas izpilde vēl nav beigusies? (diskusija)

Konveijera izmantošana





Konveijera posmi

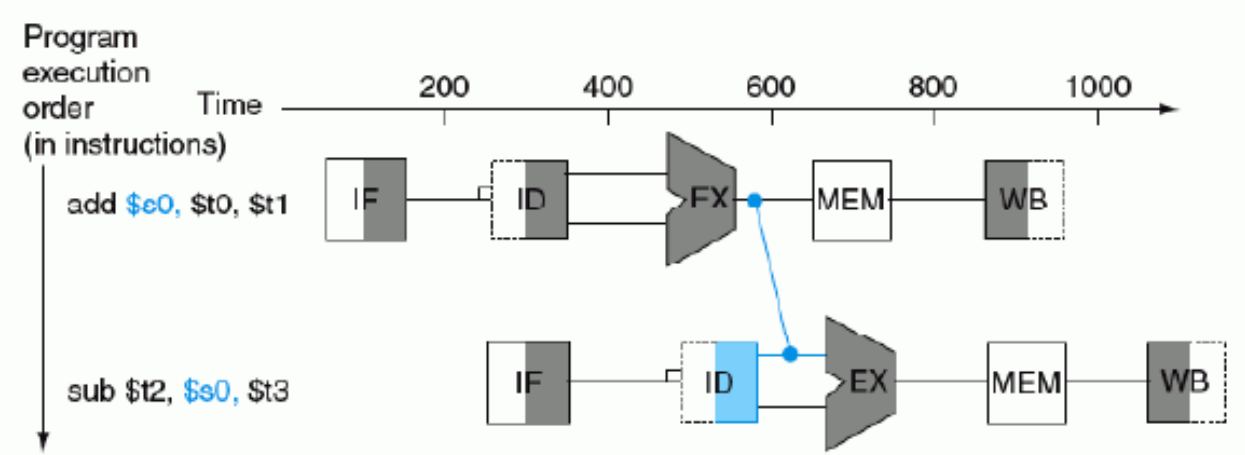
Instr. No.	Pipeline Stage						
1	IF	ID	EX	MEM	WB		
2		IF	ID	EX	MEM	WB	
3			IF	ID	EX	MEM	WB
4				IF	ID	EX	MEM
5					IF	ID	EX
Clock Cycle	1	2	3	4	5	6	7

Konveijera *briesmas* (*hazards*)

- Noteiktas situācijas, kad nākošā instrukcija nevar izpildīties nākošajā takts ciklā:
  - Strukturālas *briesmas* – *dzelži* nespēj izpildīt šo instrukciju kombināciju vienā takts ciklā (piemēram, vienlaicīga *vēršanās* pie atmiņas)
  - Datu *briesmas* – kādam *solim* nepieciešams pagaidīt, lai pabeidz izpildi cits *solis*
  - Kontroles *briesmas* – jāizpilda zarošanās, bet vēl nav zināms rezultāts, uz kura bāzes tas jādara

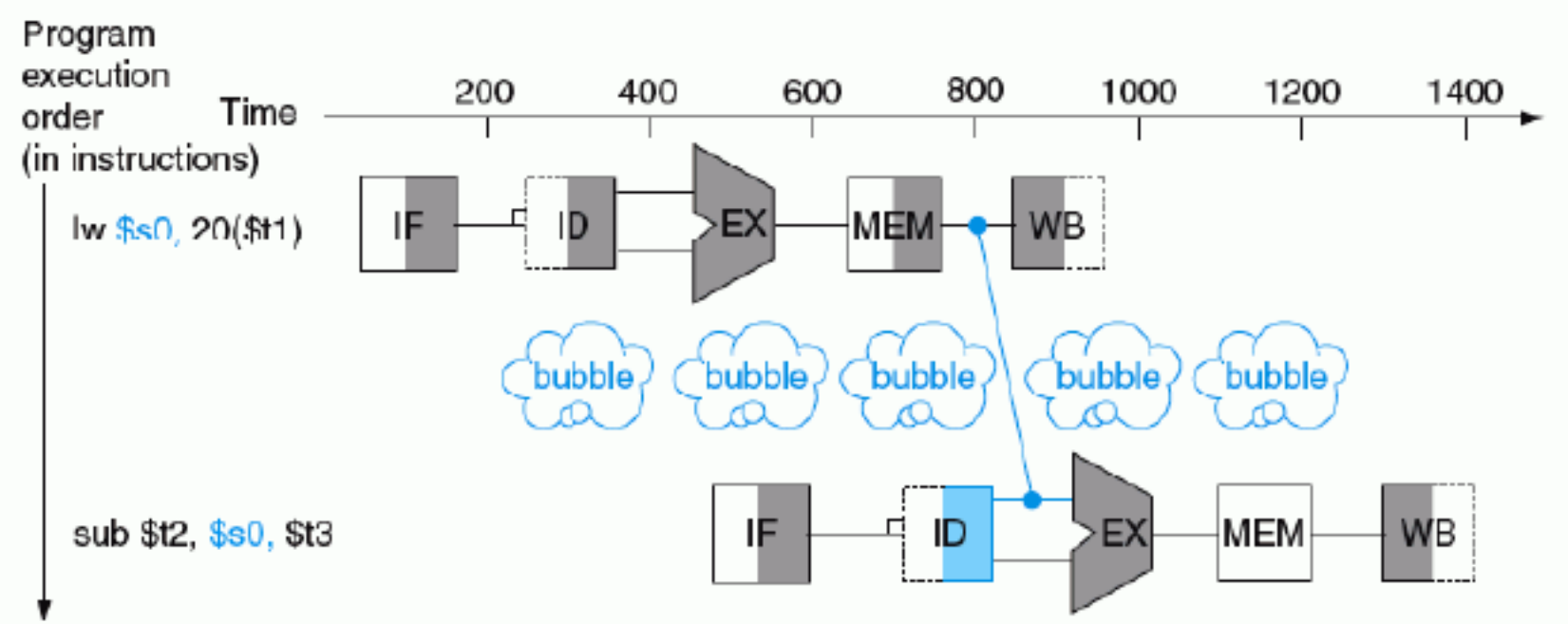
Datu *briesmas* I

- Var cerēt, ka kompilators ņems vērā potenciālās problēmas, un kaut ko darīs lietas labā, bet ne vienmēr...
- Iespējamie risinājumi
  - Izmantojot papildus dzelžus, paņemt datus no iekšējiem buferiem, negaidot, kamēr tie nokļūs atmiņā vai reģistros (*forwarding/bypassing*)



Datu *briesmas* II

- Jāņem vērā, ka laika mašīnas mums nav (ja dati vēl nav pat sarēķināti, tos tālāk padot nevarēs, nāksies vien gaidīt...)

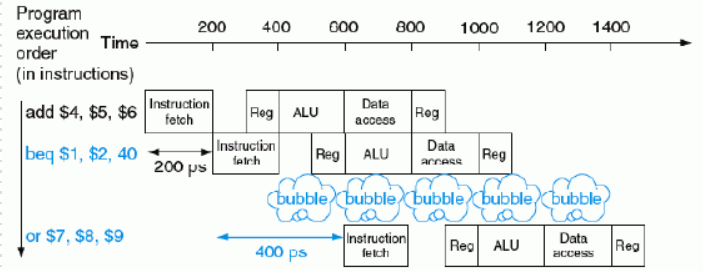


Datu *briesmas* III

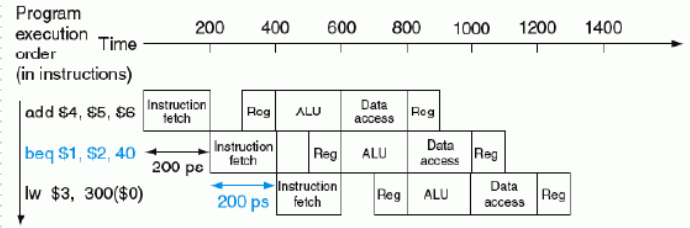
- **Read after Write (RAW)**
  - Dati tiek ierakstīti un uzreiz nolasīti
  - Ja pirmā instrukcija vēl nav pabeigusies, otrā var nolasīt nekorektu rezultātu
- **Write after Read (WAR)**
  - Dati tiek nolasīti un uzreiz ierakstīti
  - Ja otrā instrukcija ir pabeigusies ātrāk nekā pirmā, pirmā var nolasīt nekorektu rezultātu
- **Write after Write (WAW)**
  - Dati tiek ierakstīti un vēlreiz ierakstīti
  - Ja otrā instrukcija ir pabeigusies ātrāk nekā pirmā, saglabātais rezultāts var būt nekorekts

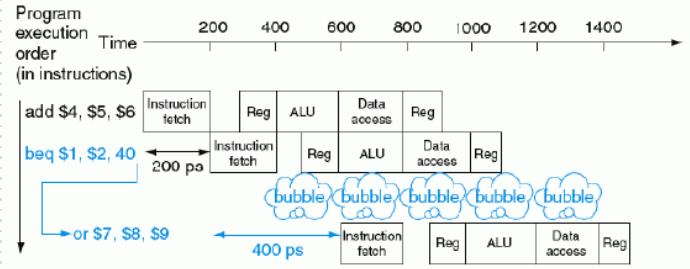
Kontroles *briesmas*

- Izpildot *branch* instrukciju, līdz noteiktam brīdim nav zināma nākošās instrukcijas adrese



- Iespējamie varianti:
  - Gaidīt, kamēr būs zināma adrese
  - *Minēt* (piemēram, cerēt, ka nekādas pārejas nebūs)
  - *Mest monētu* (dažos gadījumos cerēt, ka pāreja nebūs, citos, ka būs)





Praktiskie darbi

- Strādājam pie kursa projekta KP3

Pateicos par uzmanību!

Jautājumi?