Latvijas Universitāte
Datorikas fakultāte

# DLX procesors un instrukcijas.

**Kurss "Ievads digitālajā projektēšanā"**

**Lekcija 05.11.2010**

Autors: **Artis Mednis**

Rīga 2010

# Vidus semestra kontroldarbs

- Rezultātu analīze
- Pareizās atbildes

# DLX procesors

- DLX procesors
  - pieder pie RISC arhitektūras
  - vienkāršots 32 bitu MIPS procesors
  - pamatā domāts dažādu datoru arhitektūras kursu apguvei
- Oriģinālais MIPS risinājums
  - visas instrukcijas tiek izpildītas vienā ciklā
- DLX risinājums
  - tiek izmantota datu pārsūtīšanas un instrukciju pārkārtošanas sistēma
  - pēc garāku instrukciju izpildes rezultāts tiek *nolikts* atbilstošajā vietā
  - *uz āru* instrukciju izpilde joprojām izskatās lineāra

# DLX instrukcijas I

- Tāpat kā MIPS procesora gadījumā eksistē 3 instrukciju klases
  - R-tipa – satur norādes uz 3 reģistriem
  - I-tipa – satur norādes uz 2 reģistriem un 16 bitu platu vērtību
  - J-tipa – satur 26 bitu platu adreses vērtību
- OPCODE
  - 6 biti, kas ļauj izmantot 64 dažādas instrukcijas
- Reģistri
  - 5 biti, kas ļauj izmantot 32 dažādus reģistrus

# DLX instrukcijas II

- Vai 64 dažādas instrukcijas ir daudz vai maz? (diskusija)
- Vai DLX procesora instrukciju formāts ļauj palielināt izpildāmo instrukciju skaitu? (diskusija)
- R-tipa instrukcijas formāts
  - OPCODE (6) + 3xREGISTER (15) = 21
  - mums ir 32 bitu vārds
  - varam izmantot bitus 5:0 *papildus* instrukcijām
  - kādus ierobežojumus tas uzliek? (diskusija)
  - pareizā atbilde: papildus instrukcijas var strādāt tikai ar reģistriem

# Reģistri

- Tāpat kā MIPS procesora gadījumā
  - 32 reģistri, katrs 32 bitu plats
  - reģistrs #0 vienmēr satur vērtību 0, kuru nevar izmainīt
  - reģistram #31 ir īpašs pielietojums, kurš atkarīgs no izpildāmās instrukcijas (sīkāk kādā no nākamajiem slaidiem)
  - speciāls reģistrs PC, kurš satur norādi uz izpildāmo instrukciju atmiņā

# Aritmētiskās/loģiskās instrukcijas

| Instr. | Description | Format | Opcode | Operation (C-style coding) |
|---|---|---|---|---|
| ADD | add | R | 0x20 | Rd = Rs1 + Rs2 |
| ADDI | add immediate | I | 0x08 | Rd = Rs1 + extend(immediate) |
| AND | and | R | 0x24 | Rd = Rs1 & Rs2 |
| ANDI | and immediate | I | 0x0c | Rd = Rs1 & immediate |
| OR | or | R | 0x25 | Rd = Rs1 \| Rs2 |
| ORI | or immediate | I | 0x0d | Rd = Rs1 \| immediate |
| SUB | subtract | R | 0x22 | Rd = Rs1 - Rs2 |
| SUBI | subtract immediate | I | 0x0a | Rd = Rs1 - extend(immediate) |
| XOR | exclusive or | R | 0x26 | Rd = Rs1 ^ Rs2 |
| XORI | exclusive or immediate | I | 0x0e | Rd = Rs1 ^ immediate |

- Atšķirība stap *immediate* un *extend(immediate)*
  - *immediate* – kreisajā pusē tiek pieliktas 16 nulles
  - *extend(immediate)* – kreisajā pusē tiek pieliktas 16 vērtības, identiskas vērtībai kreisajā malējā pozīcijā

# Nobīdes instrukcijas

| Instr. | Description | Format | Opcode | Operation (C-style coding) |
|---|---|---|---|---|
| SLL | shift left logical | R | 0x04 | Rd = Rs1 << (Rs2 % 8) |
| SLLI | shift left logical immediate | I | 0x14 | Rd = Rs1 << (immediate % 8) |
| SRA | shift right arithmetic | R | 0x07 | as SRL & see below |
| SRAI | shift right arithmetic immediate | I | 0x17 | as SRLI & see below |
| SRL | shift right logical | R | 0x06 | Rd = Rs1 >> (Rs2 % 8) |
| SRLI | shift right logical immediate | I | 0x16 | Rd = Rs1 >> (immediate % 8) |

- Atšķirība stap *SRL* un *SRA*
  - SRL – no kreisās puses tiek *iebīdītas* nulles
  - SRA – no kreisās puses tiek *iebīdīta* vērtība, kas identiska vērtībai kreisajā malējā pozīcijā
- Ko dod izteiksme % 8 ? (diskusija)

# Nosacījuma pārbaudes instrukcijas

| Instr. | Description | Format | Opcode | Operation (C-style coding) |
|---|---|---|---|---|
| SEQ | set if equal | R | 0x28 | Rd = (Rs1 == Rs2 ? 1 : 0) |
| SEQI | set if equal to immediate | I | 0x18 | Rd = (Rs1 == extend(immediate) ? 1 : 0) |
| SLE | set if less than or equal | R | 0x2c | Rd = (Rs1 <= Rs2 ? 1 : 0) |
| SLEI | set if less than or equal to immediate | I | 0x1c | Rd = (Rs1 <= extend(immediate) ? 1 : 0) |
| SLT | set if less than | R | 0x2a | Rd = (Rs1 < Rs2 ? 1 : 0) |
| SLTI | set if less than immediate | I | 0x1a | Rd = (Rs1 < extend(immediate) ? 1 : 0) |
| SNE | set if not equal | R | 0x29 | Rd = (Rs1 != Rs2 ? 1 : 0) |
| SNEI | set if not equal to immediate | I | 0x19 | Rd = (Rs1 != extend(immediate) ? 1 : 0) |

# Pārejas instrukcijas

| Instr. | Description | Format | Opcode | Operation (C-style coding) |
|---|---|---|---|---|
| BEQZ | branch if equal to zero | I | 0x04 | PC += (Rs1 == 0 ? extend(immediate) : 0) |
| BNEZ | branch if not equal to zero | I | 0x05 | PC += (Rs1 != 0 ? extend(immediate) : 0) |
| J | jump | J | 0x02 | PC += extend(value) |
| JAL | jump and link | J | 0x03 | R31 = PC + 4 ; PC += extend(value) |
| JALR | jump and link register | I | 0x13 | R31 = PC + 4 ; PC = Rs1 |
| JR | jump register | I | 0x12 | PC = Rs1 |

- *JALR* un *JR* gadījumā tiek izmantota tikai reģistra *Rs1* vērtība, otra reģistra *Rs2* un *immediate* vērtības tiek *izmestas*
- *JAL* un *JALR* gadījumā parādās reģistrs #31 – ko tas varētu darīt? (diskusija)
- Pareizā atbilde: adrese, uz kuru atgriezties

# *Load/Store* instrukcijas

| Instr. | Description | Format | Opcode | Operation (C-style coding) |
|---|---|---|---|---|
| LHI | load high bits | I | 0x0f | Rd = immediate << 16 |
| LW | load woRd | I | 0x23 | Rd = MEM[Rs1 + extend(immediate)] |
| SW | store woRd | I | 0x2b | MEM[Rs1 + extend(immediate)] = Rd |

- *SW* instrukcijas gadījumā reģistra *Rs1* saturs tiek izmantots adreses aprēķinam, bet reģistra *Rd* saturs nonāk izrēķinātajā atmiņas adresē
- *LHI* instrukcija tiek izmantota, lai divos soļos ielādētu 32 bitu konstantes:
  - LHI R1,#0x1234
  - ORI R1,R1,#0x5678

# Ārējie signāli

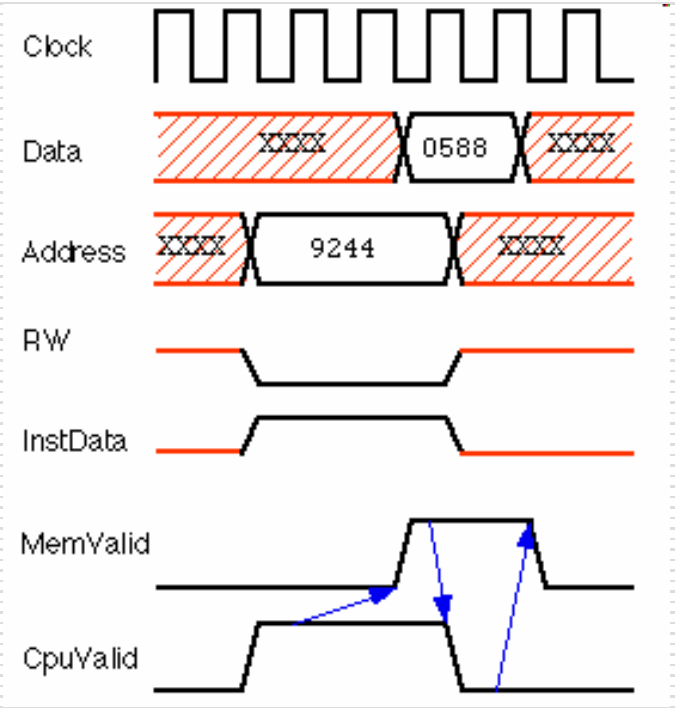
- *RESET*
  - ieslēdzot procesoru, ir jāzin, ar ko tas sāks savu darbu
  - reizēm ir nepieciešams jau darbojošos procesoru nolikt noteiktā sākuma stāvoklī
  - uzstādot *RESET* stāvoklī 1, reģistrā *PC* tiek ierakstīta vērtība 0
  - uzstādot *RESET* stāvoklī 0, tiek sākta programmas izpilde no adreses 0
- Clock
  - *debugging* režīmā varētu tikt izmantots manuāls *Clock* signāls
  - darba režīmā izmantojams reāls *Clock* signāls ar frekvenci, kura ir pietiekami zema korektai procesora darbībai

# Atmiņas saskarne

- **Address[31:0]** – atmiņas adrese, kura tiks lasīta vai rakstīta
- **Data[31:0]** – dati, kurus vai nu lasīs no atmiņas, vai arī rakstīs atmiņā
- **InstData** – instrukcijas nolasīšana (0) vai datu lasīšana/rakstīšana (1)
- **RW** – atmiņas lasīšana (0) vai rakstīšana (1)
- **CpuValid** – procesors uzstāda 1, kad ir sagatavojis stabilus signālus {Address, Data (rakstīšanas gadījumā), Instdata un RW}
- **MemValid** – atmiņa uzstāda 1, kad ir sagatavojusi stabilu signālu Data (lasīšanas gadījumā)

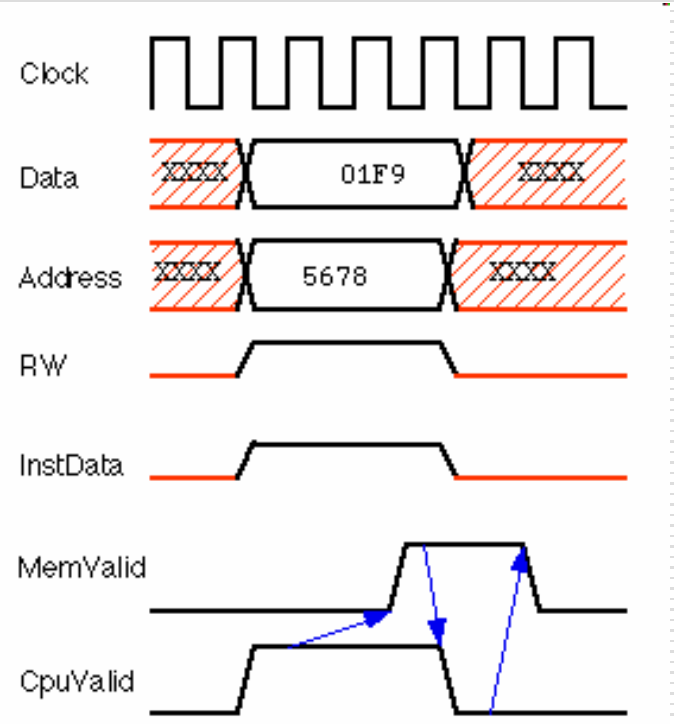
# Atmiņas lasīšana

- attiecas gan uz instrukcijas, gan datu nolasīšanu
- procesors uzstāda signālus {Address, Instdata un RW}, kuriem ir jābūt stabiliem pie katras CLK uzlecošās sķautnes, kamēr vien CpuValid == 1
- kad MemValid tiek uzstādīts par 1, pie kārtējās CLK uzlecošās šķautnes tiek nolasīta Data vērtība un CpuValid tiek uzstādīts par 0
- tiklīdz CpuValid == 0, pie nākošās uzlecošās CLK šķautnes MemValid tiek uzstādīts par 0



# Atmiņas rakstīšana

- Atšķirības no atmiņas lasīšanas:
  - RW == 1
  - InstData rakstot vienmēr ir 1, jo instrukcijas atmiņā netiek rakstītas



# Praktiskie darbi

- Turpinām strādāt pie kursa projekta KP2

Pateicos par uzmanību!

Jautājumi?