Rechnerarchitektur

Marián Vajteršic und Helmut A. Mayer

Fachbereich Computerwissenschaften
Universität Salzburg
marian@cosy.sbg.ac.at und helmut@cosy.sbg.ac.at
Tel.: 8044-6344 und 8044-6315

3. Mai 2017

Die DLX-Maschine

Die DLX-Maschine

Zielsetzungen für einen konkreten Prozessor (aufgrund der Überlegungen aus dem vorigen Kapitel)

- Einfache Load-Store-Architektur
- Befehlscode fester Länge (Befehlssatz für effiziente Pipeline)
- Befehlssatz für effiziente Compiler

→ Eine **hypothetische** (nicht kommerziell erhältliche) **Maschine**, die **einen repräsentativen Durchschnitt** vieler realer Prozessoren darstellt:

AMD 29K, DEC station 3100, HP 850, IBM 801, Intel i860, MIPS M/120A, MIPS M/1000, Motorola 88K, RISC I, SGI 4D/60, SPARC station-1, SUN-4/110, SUN-4/260

Alle Zahlen addieren und Durchschnitt bilden:
$560 = DLX$

Die DLX-Maschine

Prozessor-Simulatoren:

- Online-DLX-Simulator, basierend auf dlxsim:
  `http://lv.cosy.sbg.ac.at/digitale/dlxwsim/`
- dlxsim: Unix-Programm mit Command-Line-Interface
- openDLX
- WinDLX: 16-Bit-Windows-Programm mit GUI

Weitere Unterlagen:

- Kurze Einführung in die Assemblerprogrammierung in DLX:
  `http://lv.cosy.sbg.ac.at/digitale/ws1011/asmskript.pdf`
- Kurzübersicht der DLX-Befehle:
  `http://lv.cosy.sbg.ac.at/digitale/ws1011/befehle.pdf`

DLX-Befehlssatz

- Register
  - DLX beinhaltet **32 32-Bit**-Register:
    GPR (General Purpose Register) bezeichnet mit `R0`, `R1`, ..., `R31`
    $R0 = 0$ (immer) (Ein Laden dieses Registers bleibt ohne Wirkung.)
  - Zusätzlich noch **32 32-Bit**-Floating-Point-Register (**FPR**) `F0`, `F1`, ..., `F31`
    (Diese können auch im 64-Bit-Modus angesprochen werden: Double Precision)
    Wenn als Double verwendet: Alle geraden Registernummern werden verwendet, wobei die ungeraden das LSWord der Zahl im Double-Format beinhalten.

    16 Double-Precision-Register-Paare:

MSW $\rightarrow$	0	2	$\cdots$	30
LSW $\rightarrow$	1	3	$\cdots$	31

DLX-Befehlssatz

- Datentypen

  Alle Datentypen: 8-64 Bit (**Byte**, **Halbwort**, **Wort**, **Doppelwort**) können adressiert werden.

  - Die Operationen der DLX arbeiten ausschließlich mit **Operanden** mit **32 Bit** (Integer oder Float) und **64 Bit (Double)**.
  - Werden Bytes oder Halbworte geladen, werden die unberührten Stellen des Registers (von links) entweder mit 0 aufgefüllt (**unsigned**) oder erweitert (vorzeichenrichtig) (**signed**).

Adressierung des DLX-Speichers

- Byte-adressierbar.
  (Wort hat 4 Bytes, also hat das nächste Wort die Adresse (neu) = Adresse (alt) + 4.)
- Ausgerichtet (aligned).
- Im Big Endian Format (höherwertiges Byte (MSB) zuerst).

- Adressierungsmodi (wird später näher spezifiziert)
- Befehlsformat (wird später näher spezifiziert)

DLX-Befehlssatz

- **Datentransfer**
  - **LB** Lade Byte signed
  - **LBU** Lade Byte Unsigned
  - **SB** Store Byte

32 Bit:	8 Bit	8 Bit	8 Bit	8 Bit
				01010011
signed:	00000000	00000000	00000000	**0**1010011
oder:	11111111	11111111	11111111	**1**1011011
unsigned:	00000000	00000000	00000000	11011011

  - `LBU R1, 100(R2)`: Hole **Byte** von $R2 + 100$ in `R1` unsigned
  - L**H**, LHU, SH: Lade oder speichere **Halbwort** signed oder unsigned
  - `LH R3, 100(R1)`: Hole Halbwort von $R1 + 100$ signed nach `R3`

DLX-Befehlssatz

- **Datentransfer** (Fortsetzung)
  - LW, SW: Lade oder speichere **Wort** (man braucht nicht signed bzw. unsigned, da alles verbraucht wird)
  - `SW 100(R2), R1`: Speichere Wort aus `R1` nach Memory-Stelle mit Adresse $R2 + 100$
  - LF, LD, SF, SD: Lädt und speichert Floats und Doubles
    (Warum braucht man eigene Befehle? Weil eigene Float-Register.)
  - `LD F2, 100(R2)`: Holt Double von $R2 + 100$ nach `F2` (MSByte) und `F3` (LSByte)
    (für Double sind nur gerade Register erlaubt)

DLX-Befehlssatz

- **Moves**
  - MOVEI2S, MOVES2I: Transferiert Daten zwischen GPRs und Spezialregister
  - MOVF, MOVD: Transferiert Daten zwischen FPRs

Beispiele (Datentransfer)

- `LW R1, 30(R2)`: Load Word signed/unsigned
  $R1 \leftarrow_{32} M[30 + R2]$
- `LW R1, 1000(R0)`: Load Word
  $R1 \leftarrow_{32} M[1000 + 0]$
- `LB R1, 40(R3)`: Load Byte (signed)
  $R1 \leftarrow_{32} (M[40 + R3]_7)^{24} \# \# M[40 + R3]$
- `LBU R1, 40(R3)`: Load Byte (unsigned)
  $R1 \leftarrow_{32} 0^{24} \# \# M[40 + R3]$
- `LH R1, 40(R3)`: Load Halfword signed
  $R1 \leftarrow_{32} (M[40 + R3]_7)^{16} \# \# M[40 + R3] \# \# M[41 + R3]$
- `LHU R1, 40(R3)`: Load Halfword unsigned
  $R1 \leftarrow_{32} 0^{16} \# \# M[40 + R3] \# \# M[41 + R3]$
- `LF F0, 50(R3)`: Load Float signed/unsigned
  $F0 \leftarrow_{32} M[50 + R3]$
- `LD F0, 50(R2)`: Load Double
  $F0 \# \# F1 \leftarrow_{64} M[50 + R2]$

Beispiele (Datentransfer)

- `SW 500(R4), R3`: Store Word signed/unsigned
  $M[500 + R4] \leftarrow_{32} R3$
- `SF 40(R3), F0`: Store Float
  $M[40 + R3] \leftarrow_{32} F0$
- `SD 40(R3), F0`: Store Double
  $M[40 + R3] \leftarrow_{32} F0$
  $M[44 + R3] \leftarrow_{32} F1$
- `SH 502(R2), R3`: Store Halfword
  $M[502 + R2] \leftarrow_{16} R3_{16\ldots32}$
- `SB 41(R3), R2`: Store Byte
  $M[41 + R3] \leftarrow_{8} R2_{24\ldots32}$

Beispiele (Datentransfer)

- `MOVF F3, F7`: Kopiere F7 nach F3
  Oder:
  `SF 100(R2), F7`: $F7 \rightarrow M[R2 + 100]$
  `LF F3, 100(R2)`: $F3 \leftarrow M[R2 + 100]$
- `MOVFP2I`, `MOVI2FP`: Datentransfer zwischen FPRs und GPRs
- `MOVFP2I R1, F2`: Kopiere F2 nach R1
  (man kopiert die Zahl in IEEE-Darstellung nach R1)

Arithmetik und Logik

- `ADD`: Addiert Register signed
- `ADDI`: Addiert Immediate signed
- `ADDU`: Addiert Register unsigned
- `ADDUI`: Addiert Immediate unsigned

`ADD R3, R2, R1`: $R1 + R2$ nach $R3$ (signed) (wenn MSB gesetzt: negative Zahl)

- `SUB`: Subtrahiert Register signed
- `SUBI`: Subtrahiert Immediate signed
- `SUBU`: Subtrahiert Register unsigned
- `SUBUI`: Subtrahiert Immediate unsigned

`SUBI R3, R2, #42`: $R2 - 42$ nach $R3$ (signed)

Arithmetik und Logik

- `MULT`, `MULTU`, `DIV`, `DIVU`:
  Multipliziert/dividiert FPRs signed, unsigned (nur Floating-Point-Multiplikationen möglich)

`MULT F3, F2, F1`: $F1 \cdot F2$ nach $F3$ (signed)

- `AND`, `ANDI`: Logisches Und (auch mit Immediate)

`ANDI, R3, R2, #16`: Bit 4 von $R2$ nach $R3$
$b_4 b_3 b_2 b_1 b_0 \equiv 10000$

- `OR`, `ORI`, `XOR`, `XORI`: Logisches OR, XOR (immediate)

Control (Steuerung)

- `BEQZ`, `BNEZ`:
  Branch, wenn GPR (General Purpose Register) gleich/ungleich 0

  `BNEZ R3, LOOP`: Springe nach „LOOP“ (Marke) wenn $R3$ ungleich 0

Alle Schleifen (While, For) sind mit diesen Anweisungen realisiert.

Beispiel (Schleife)

(Die Speicherzellen 100-199 werden mit 0 gefüllt.)

	`ADD R1, R0, #100`	$R1 = 100$
`loop:`	`SUB R1, R1, #4`	$R1$ erniedrigen
		(um 4 Bytes = 1 Word)
	`SW 100(R1), R0`	$M[100 + R1] \leftarrow {}_{32}0$
	`BNEZ R1, LOOP`	Sprung solange $R1 \neq 0$

1. Durchlauf: Speicherzellen $100 + 96$, 197, 198, 199 mit 0 gefüllt
2. Durchlauf: Speicherzellen $100 + 92$, 193, 194, 195 mit 0 gefüllt

...

25. Durchlauf: Zellen $100 + 0$, 101, 102, 103 mit 0 gefüllt

$R1 = 0 \rightarrow$ kein Sprung nach `LOOP`, Programm wird fortgesetzt

Control (Steuerung)

- `BFPT`, `BFPF`: Branch entsprechend Compare Bit in FP (Statusregister) (FPSR)

  `BFPT LOOP`: Springe nach Marke `LOOP`, wenn FPSR true (wenn FP-Flag gesetzt: Springe nach `LOOP`)

- `J`, `JR`: Sprung absolut (26 Bit) oder **Register**

  `JR R1`: Springe nach Adresse in $R1$

- `JAL` (Jump And Link), `JALR` (Jump And Link Register):
  Sprung, speichere $PC + 4$ in $R31$ (Link)

  `JAL PROC`: Rufe Prozedur `PROC`, Rücksprung-Adresse in $R31$

Beispiele

- `J NAME`: Jump
  $PC \leftarrow$ `NAME`, wobei
  $((PC + 4) - 2^{25}) \leq$ `NAME` $< ((PC + 4) + 2^{25})$
- `JAL NAME`: Jump And Link
  $R31 \leftarrow PC + 4$; $PC \leftarrow$ `NAME`, wobei
  $((PC + 4) - 2^{25}) \leq$ `NAME` $< ((PC + 4) + 2^{25})$
- `JALR R2`: Jump And Link Register
  $R31 \leftarrow PC + 4$; $PC \leftarrow R2$
- `JR R3`: Jump Register
  $PC \leftarrow R3$
- `BEQZ R4, NAME`: Branch Equal Zero
  `IF(R4=0)` $PC \leftarrow$ `NAME`, wobei
  $((PC + 4) - 2^{25}) \leq$ `NAME` $< ((PC + 4) + 2^{25})$
- `BNEZ R4, NAME`: Branch Not Equal Zero
  `IF(R4!=0)` $PC \leftarrow$ `NAME`, wobei
  $((PC + 4) - 2^{25}) \leq$ `NAME` $< ((PC + 4) + 2^{25})$

Control (Steuerung)

Bei allen direkten Sprüngen wird die Zieladresse durch einen relativen Offset zum Befehlszähler mit einer Wortbreite von 16 oder 26 Bit kodiert.

- Dadurch sind direkte Sprünge außerhalb eines Radius von $\pm 2^{15}$ bzw. $\pm 2^{25}$ nicht möglich.
- Für Sprünge außerhalb dieses Radius bzw. für absolute Sprünge muss ein indirekter Sprungbefehl (`JR`) verwendet werden.

Schiebeoperationen

- `SLL`, `SRL`, `SLLI`, `SRLI`, `SRA`, `SRAI`: Logisches, arithmetisches Schieben (`I`: Immediate)

  `SLLI R3, #3`: Schiebe $R3$ um 3 Stellen nach links, 0 von rechts

  `SRA R3, R2`: Verschiebe $R3$ um $R2$ nach rechts, Vorzeichen von links

- `S__`, `S__I`: Prüfen einer Bedingung `LT`, `GT`, `LE`, `GE`, `EQ`, `NE`

  `SEQ R3, R2, R1`: Setze $R3$, wenn $R1 = R2$ (wenn gleich: 1, wenn ungleich: 0)

  `SLTI R3, R2, #10`: Setze $R3$, wenn $R2 < 10$

Shift-Operationen (SRL, SLL, SRA, ~~SLA~~)

- Beim Arithmetischen Shift (A):
  Die hereingeschobenen Stellen werden mit dem Vorzeichenbit aufgefüllt.
  Gilt nur für Schieben nach rechts (weil Schieben des Vorzeichens nach links keinen Sinn hat). Deswegen ist SLA **in DLX nicht definiert**.
- Beim Logischen Shift (L):
  Die hereingeschobenen Stellen werden stets mit 0 aufgefüllt.

SRL (Shift Right Logical)

Es gilt für x positiv:

$$\texttt{SRL}(x) = \left\lfloor \frac{x}{2} \right\rfloor$$

($\left\lfloor \frac{x}{2} \right\rfloor$ bezeichnet die nächst-kleinere ganze Zahl zum Bruch $\frac{x}{2}$).

- Beispiel:
  - $(x_{(1)} = x_{(2)}) = x = 0111 = 7$
    $\texttt{SRL}(x) = 0011 = 3 = \left\lfloor \frac{7}{2} \right\rfloor = \lfloor 3.5 \rfloor$
- Bemerkung: Für negative Zahlen gilt das nicht:
  - $x_{(1)} = -1 = 1110$
    $\texttt{SRL}(x_{(1)}) = 0111 = 7 \neq \left\lfloor -\frac{1}{2} \right\rfloor = \lfloor -0.5 \rfloor = -1$
  - $x_{(2)} = -2 = 1110$
    $\texttt{SRL}(x_{(2)}) = 0111 = 7 \neq \left\lfloor -\frac{2}{2} \right\rfloor = -1$

SLL (Shift Left Logical)

Es sei x in n-Bit-Darstellung. Es gilt für

- x positiv: $x < 2^{n-2}$
- $x_{(2)}$ negativ: $x \geq -2^{n-2}$

$$\mathtt{SLL}(x) = 2x\,.$$

- Beispiel: $n = 4$
  - $(x_{(1)} = x_{(2)}) = x = 0011 = 3 (< 2^{4-2} = 4)$
    $\mathtt{SLL}(x) = 0110 = 6 (= 2 \cdot 3)$
  - $x_{(2)} = 1100 = -4 (= -8 + 4)$, $(-4 = -2^{4-2} = -4)$
    $\mathtt{SLL}(x) = 1000 = -8 (= 2 \cdot (-4))$
  - $x_{(2)} = 1111 = -1 (= -8 + 4 + 2 + 1)$, $(-1 > -2^{4-2} = -4)$
    $\mathtt{SLL}(x) = 1110 = -2 (= 2 \cdot (-1))$

SLL (Shift Left Logical)

- Bemerkung 1: Es gilt nicht für $x_{(1)}$
  - $x_{(1)} = 1001 = -6$
    $\texttt{SLL}(x) = 0010 = 2 \neq 2 \cdot (-6)$
  - $x_{(1)} = 1100 = -3$
    $\texttt{SLL}(x) = 1000 = -7 \neq 2 \cdot (-3)$
- Bemerkung 2: Es gilt nicht für $x_{(2)}$, wobei $x < -2^{n-2}$
  - $x_{(2)} = 1011 = -5$, $(-5 < -2^{4-2} = -4)$
    $\texttt{SLL}(x) = 0110 = 6 \neq 2 \cdot (-5)$
- Bemerkung 3: Es gilt nicht für $x \geq 2^{n-2}$
  - $x = 0100 = 4$, $(4 = 2^{4-2} = 4)$
    $\texttt{SLL}(x) = 1000 = \begin{matrix} -8 \text{ (2-Komplement)} \\ -7 \text{ (1-Komplement)} \end{matrix} \neq 2 \cdot 4$
  - $x = 0111 = 7$, $(7 > 4)$
    $\texttt{SLL}(x) = 1110 = \begin{matrix} -2 \text{ (2-Komplement)} \\ -1 \text{ (1-Komplement)} \end{matrix} \neq 2 \cdot 7 = 14$

SRA (Shift Right Arithmetical)

1. Für x positiv oder negativ (im 2-Komplement) gilt:

$$\texttt{SRA}(x) = \left\lfloor \frac{x}{2} \right\rfloor$$

($\left\lfloor \frac{x}{2} \right\rfloor$ bezeichnet die nächst-kleinere ganze Zahl zum Bruch $\frac{x}{2}$).

2. Für x negativ (im 1-Komplement) gilt:

$$\texttt{SRA}(x) = \left\lceil \frac{x}{2} \right\rceil$$

($\left\lceil \frac{x}{2} \right\rceil$ bezeichnet die nächst-größere ganze Zahl zum Bruch $\frac{x}{2}$).

SRA (Shift Right Arithmetical)

Beispiel:

1.
   - $x = 0111 = 7$
     $\texttt{SRA}(x) = 0011 = 3 = \lfloor \frac{7}{2} \rfloor = \lfloor 3.5 \rfloor$
   - $x = 0010 = 2$
     $\texttt{SRA}(x) = 0001 = 1 = \lfloor \frac{2}{2} \rfloor = \lfloor 1 \rfloor$
   - $x_{(2)} = 1111 = -1$
     $\texttt{SRA}(x) = 1111 = -1 = \lfloor -\frac{1}{2} \rfloor = \lfloor -0.5 \rfloor$
   - $x_{(2)} = 1100 = -4$
     $\texttt{SRA}(x) = 1110 = -2 = \lfloor -\frac{4}{2} \rfloor = \lfloor -2 \rfloor$

2.
   - $x_{(1)} = 1000 = -7$
     $\texttt{SRA}(x) = 1100 = -7 + 4 = -3 = \lceil -\frac{7}{2} \rceil = \lceil -3.5 \rceil$
   - $x_{(1)} = 1101 = -2$
     $\texttt{SRA}(x) = 1110 = -7 + 4 + 2 = -1 = \lceil -\frac{2}{2} \rceil = \lceil -1 \rceil$

Floating Point

- `ADDF`, `ADDD`: Addiere Floats, Doubles
  `ADDD F8, F4, F6`: Addiere Doubles in F4 und F6 nach F8
- `SUBF`, `SUBD`: Subtrahiere Floats, Doubles
- `MULTF`, `MULTD`: Multipliziere Floats, Doubles
- `DIVF`, `DIVD`: Dividiere Floats, Doubles
- `CVTx2y`: Konvertiere Integer, Float, Double
  `CVTF2I F2, F3`: Wandle Float in `F3` zu Integer in `F2`
- `__F`, `__D`: Vergleiche Floats, Doubles `LT`, `GT`, `LE`, `GE`, `EQ`, `NE`
  `GED F4, F6`: **Compare Bit** im **Floating Point Status Register** wird gesetzt, wenn $F4 \geq F6$ ($F4$, $F6$: Doubles)

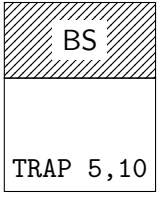
Floating Point: Besonderheiten

- Die Multiplikation und Division wird (ausnahmslos) von der Floating-Point-Unit durchgeführt $\rightarrow$ die Zahlen müssen in FPRs stehen.
- Die Rücksprungadresse eines Prozeduraufrufs wird mit dem Befehl `JAL` (immer) in $R31$ abgelegt.
- Es gibt im Befehlssatz der DLX nicht explizit:
  1. Laden einer Konstanten in ein Register
  2. Verschieben eines Datums von einem GPR in ein anderes (Register Move).

  Beide Befehle kann man implizit mit dem Nullregister $R0$ ausführen:
  1. `ADDI R1, R0, #5`: Lade $R1$ mit 5
     d. h. $R1 \leftarrow 0 + 5 = 5$
  2. `ADD R1, R0, R2`: Kopiere (Move) $R2$ nach $R1$
     d. h. $R1 \leftarrow 0 + R2 = R2$

Trap

Trap: Jump, der immer auf (systemabhängige) vordefinierte Stelle im Speicher springt (Betriebssystem)



$\rightarrow$ springe an Stelle 5 im Betriebssystem und führe Anweisung 10 aus

Beispiel

Betrag in ATS: x
Betrag in EUR: y
Wechselweise umwandeln, 2 Speicherstellen

```
(WORD) ATS_BETRAG
(WORD) EUR_BETRAG
(WORD) FACTOR
```

Mögliche Speicherstellen haben symbolische Namen (Aufgabe `EUR_BETRAG` in Adresse umzuwandeln übernimmt Assembler)

`LF`	`F1, EUR_BETRAG`	Load Inhalt der Adresse `EUR_BETRAG` nach *F1*
`LF`	`F3, FACTOR`	Load Inhalt der Adresse `FACTOR` nach *F3*
`MULTF`	`F2, F1, F3`	Berechne ATS
`SF`	`ATS_BETRAG, F2`	Store *F2* in die Adresse `ATS_BETRAG` im Speicher

Integer Benchmark

Integer Benchmark-Messungen
(Benchmark: Typische Programme, mit denen die Performance gemessen wird)

Integer Benchmark:
Nur ALU-Befehle, z. B. Primzahlen suchen, Textverarbeitungsprogramm,
Character-/Stringverarbeitung (fast immer mit ALU/Integer) ergaben,

dass für die DLX-Maschine 80% aller Befehle von nur **fünf** Befehlen abgedeckt wird:
`LOAD` (26%), `BRANCH` (17%), `ADD` (14%), `COMPARE` (14%), `STORE` (9%).

Adressierungsmodi in DLX

Die einzelnen generischen (G) Adressierungsmodi (Kapitel: Rechnerarchitektur) werden in DLX (D) mit Hilfe von `LW` (Load Word), `ADD` (Addition) und `ADDI` (Addition Immediate) Befehlen veranschaulicht:

- Direkte Adressierung

  G: `MOVE R1, 100`
  (Hole Inhalt der absoluten Adresse 100 nach *R1*)

  D: `LW R1, 100(R0)`
  (Hole Wort von Adresse 100 nach *R1*)

Adressierungsmodi in DLX

- Indirekte Adressierung
  - Memory Deferred:

    G: `MOVE R1, (1000)`
    (Hole Inhalt der Adresse, die auf Adresse 1000 steht, nach *R1*)

    D: (Über zwei Befehle gemacht)
    `LW R5, 1000(R0)`
    `LW R1, 0(R5)`
    (Hole Wort von Adresse 1000 nach *R5* und danach hole Wort aus der Adresse in *R5* nach *R1*)

  - Register Deferred:

    G: `MOVE R1, (R2)`
    (Hole Inhalt der Adresse, die in *R2* steht, nach *R1*)

    D: `LW R1, 0(R2)`
    (Hole Wert aus der Adresse *R2* nach *R1*)

Adressierungsmodi in DLX

- Indirekte Adressierung (Fortsetzung)
  - Displacement:

    G: `MOVE R1, 100(R2)`
    (Hole Inhalt der Adresse $R2 + 100$ nach $R1$)

    D: `LW R1, 100(R2)`
    (Hole Wert aus der Adresse $100+R2$ nach $R1$)

  - Indexed:

    G: `MOVE R1, (R2+R3)`
    (Hole Inhalt der Adresse $R2 + R3$ nach $R1$)

    D: (Über zwei Befehle gemacht)
    `ADD R4, R2, R3`
    `LW R1, 0(R4)`
    (Speichere den Wert von $R2+R3$ in $R4$ und danach hole den Wert aus der Speicherstelle, die in $R4$ abgelegt ist, nach $R1$)

Adressierungsmodi in DLX

- Immediate Adressierung

  G: `MOVE R1, #100`
  (Lade *R1* mit der Zahl 100)

  D: `ADDI R1, R0, #100`
  (Addiere *R0* mit 100 und schreibe das Ergebnis (100) in *R1*)

Format von Befehlen des DLX-Befehlssatzes

Es gibt drei Typen von DLX-Befehlen: I, R, J

- I-Typ

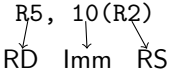
Befehlsformat:	6 Bit	5 Bit	5 Bit	16 Bit
	Opcode	RS	RD	Immediate

I-Typ codiert:

- `LOAD`
  (RD wird mit dem Wert der Speicherstelle RS + Immediate geladen)

  $RD \leftarrow MEM[RS + Immediate]$

  Beispiel: `LW R5, 10(R2)`

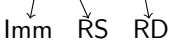
  

Format von Befehlen des DLX-Befehlssatzes

- I-Typ (Fortsetzung)
  - `STORE`
    (Inhalt des RD-Registers wird in der Speicherstelle RS + Immediate gespeichert)

    $MEM[RS + Immediate] \leftarrow RD$

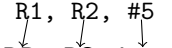
    Beispiel: `SW 10(R3), R1`

  - Alle Immediate-Befehle
    (Die Operanden in RS und Immediate werden durch die Operation in Opcode zusammen verknüpft und das Ergebnis in RD abgelegt)

    $RD \leftarrow RS\ OP\ Immediate$

    Beispiel: `ADDI R1, R2, #5`

Format von Befehlen des DLX-Befehlssatzes

- I-Typ (Fortsetzung)
  - Conditional Branch Befehle
    (RS: Register, RD: nicht benutzt)
    (Wenn die Branchbedingung *COND* für RS erfüllt ist, dann wird auf die Adresse PC + Immediate gesprungen)

    `IF` $COND(RS)$ `THEN` $PC \leftarrow MEM[PC + Immediate]$

    Beispiel: `BEQZ R1, 100`

    COND RS Imm

Format von Befehlen des DLX-Befehlssatzes

- I-Typ (Fortsetzung)
  - `JR` (Jump Register) und `JALR` (Jump And Link Register)
    (RD = 0, RS-Register, Immediate = 0)
    - `JR`
      (Im Register RS ist die Adresse, zu welcher gesprungen wird)

      $PC \leftarrow MEM[RS]$

      Beispiel: `JR R31`

    - `JALR`
      (Speichere die Rücksprungadresse $PC + 4$ in *R31* und springe nach Adresse in RS)

      $R31 \leftarrow PC + 4 \quad PC \leftarrow MEM[RS]$

      Beispiel: `JALR R2`

Format von Befehlen des DLX-Befehlssatzes

- R-Typ

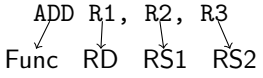
Befehlsformat:	6 Bit	5 Bit	5 Bit	5 Bit	11 Bit
	Opcode	RS1	RS2	RD	Func

R-Typ codiert:

- Register-Register ALU-Befehle
  (Die Operanden stehen in Quellregistern RS1 und RS2 und das Ergebnis der Operation Func wird in RD abgelegt)

  $RD \leftarrow RS1\ FUNC\ RS2$

  Beispiel:



Format von Befehlen des DLX-Befehlssatzes

- R-Typ (Fortsetzung):
  - `MOVxxx` Befehle
    (Kopiere RS1-Register in RD-Register)

    $RD \leftarrow RS1$

    Beispiel: `MOVF F2, F4`

    RD RS1

Format von Befehlen des DLX-Befehlssatzes

- J-Typ

Befehlsformat:	6 Bit	26 Bit
	Opcode	PC Offset

J-Typ codiert:

- `J` (Jump) und `JAL` (Jump And Link)
  - `J`
    (Sprung zur Adresse PC + Offset (26 Bit für Codierung von Offset vorhanden)

    $PC \leftarrow PC + Offset$

    Beispiel: `J TARGET` ($= PC + Offset$)
  - `JAL`
    (Speichere die Rücksprungadresse $PC + 4$ in *R31* und springe zur Adresse $PC + Offset$)

    Beispiel: `JAL TARGET` ($= PC + Offset$)

Format von Befehlen des DLX-Befehlssatzes

- J-Typ (Fortsetzung):
  - `TRAP`
    (Sprung zu einer Betriebssystem-Routine)