The DLX ISA

05-1

Coverage

Textbook Section 2.8

Topics

DLX Goals

DLX Instruction Highlights

DLX Instruction Coding

Synthetic Instructions (NIB)

05-2

DLX Design Considerations

DLX Goals

A typical RISC processor.

Incorporate features with demonstrated usefulness.

Enable simple, high-speed, implementation

Demonstration of Usefulness of Features

Covered in Chapters 1, 2.

Determined by analyzing existing ISAs.

Some usefulness illustrated with graphs (e.g., immediate sizes).

Less useful, "it would be nice," features omitted.

05-3

DLX's Useful Features

Lots of general purpose registers.

Integer and floating-point operands.

Basic arithmetic and logical operations.

Basic addressing modes: register, immediate, displacement.

Adequate immediate and displacement sizes.

Etc.

Simple, High-Speed Implementation

Load-Store Architecture: ALU instructions do not access memory.

Simple Coding: uniform instruction sizes, few instruction types.

Work Balance: Instructions do about the same amount of work.

Separate integer and FP register files.

05-4

Simple Coding Advantages

Simpler and faster decoding logic.

Execution can start before decoding complete.

Work Balance Advantages

Efficient use of CPU hardware.

Integer operations are balanced.

Floating-point operations are not. (Division takes longer than ADD.)

05-5

Separate Register File Advantages

Double the number of registers with only 1 bit per instruction (in opcode).

(Otherwise, 1 extra bit per operand would be needed.)

Splits register reads and writes between two register files.

With one large set of registers ...
... if n instructions start at once, need to access $2n$ registers ...
... all stored in one file (memory device) — expensive and slow.

With separate integer and FP register files ...
... each file would only have to provide n registers ...
... (assuming equal number of integer and FP instructions).

Note: Currently, n varies from 2 to 6.

Details on these implementation factors covered later.

05-6

DLX Registers and Data Types

Data Types

Integer: 8-bit (byte), 16-bit (half words), 32-bit (words).

Floating point: 32-bit single precision, 64-bit double precision.

Registers

32 32-bit general-purpose registers (GPRs), `r0-r31`, including a zero register, `r0`.

32 32-bit floating-point registers (FPRs), `f0,f1,f2,...,f31` ...

... which can be used as 16 64-bit registers, `f0,f2,f4,...,f30`.

A floating-point status register (for outcome of comparisons, used by branches).

05-6

DLX Instruction Highlights

05-7

Instruction Highlights

Single, but flexible, memory addressing mode: Displacement.

Special load high (LHI) instruction for (part of) 32-bit constants.

Dummy, but very handy, register r0. (Value always 0.)

Displacement Addressing Flexibility

Classic Displacement Addressing

```
lw r1, 4(r2)      ! r1 = MEM[ r2 + 4 ]
```

Register Indirect (Use zero displacement.)

```
Lw r1, 0(r2)      ! r1 = MEM[ r2 ]
```

Absolute (Use r0, limited because of immediate size.)

```
Lw r1, 1234(r0)   ! r1 = MEM[ 1234 ]
```

05-8

LHI Examples

Used to load constants.

Needed because immediate size limited to 16 bits.

Example, set r1 = 0x12345678

```
lhi r1, 0x1234          ! r1 = 0x12340000
ori r1, r1, #0x5678     ! r1 = r1  0x5678 (bitwise or)
```

05-9

Fun with `r0`, and other tricks.

Set a register to zero:

```
add  r1, r0, r0   ! r1 = 0 + 0     = 0.
addi r1, r0, #0   ! r1 = 0 + 0     = 0.
sub  r1, r1, r1   ! r1 = r1 - r1   = 0.
xor  r1, r1, r1   ! r1 = r1 xor r1 = 0.
```

Move one register to another:

```
add  r2, r1, r0   ! r2 = r1 + 0  = r1
and  r2, r1, r1   ! r2 = r1 & r1 = r1
andi r2, r1, #0   ! r2 = r1 & 0  = r1
```

Bitwise Negation

```
XORI r2, r1, #-1  ! r2 =  r1
```


EE 4720 Lecture Transparency. Formatted 11:41, 25 August 2000 from lsli05.
05-9

DLX Instruction Coding

05-10

All instructions have 6-bit opcode.

Three types.

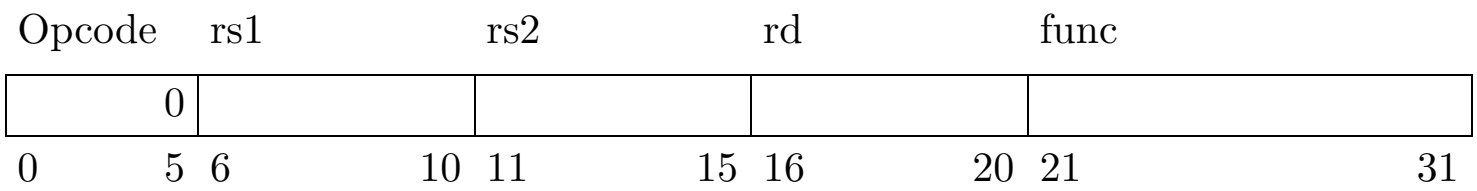
Type R: Three registers, plus extra opcode field.

Type I: Two registers, plus 16-bit immediate field.

Type J: One 26-bit immediate field.

05-11

Type R



For Type R instructions opcode field zero ...
... operation specified using func field.

Used for arithmetic, logical, and move instructions.

Sometimes just two registers used, but func field needed for operation.

Examples

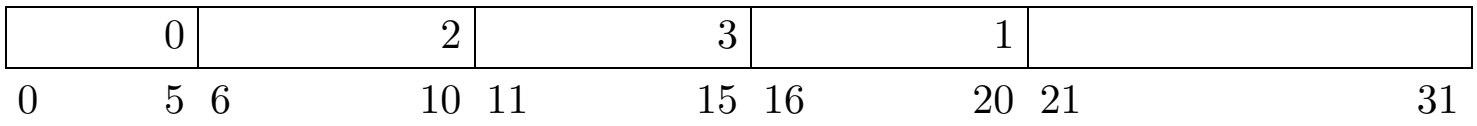
```
add    r1, r2, r3   ! Add integer registers.
addf   f1, f2, f3   ! Add FP registers, single precision.
addd   f2, f4, f6   ! Add FP registers, double precision.
movi2s f1, r1       ! Move r1 to f1. (rs2 operand not used.)
```


EE 4720 Lecture Transparency. Formatted 11:41, 25 August 2000 from lsli05.
05-11

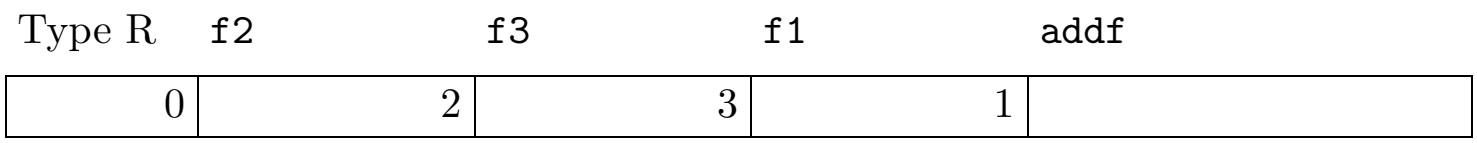
05-12

Type R Examples

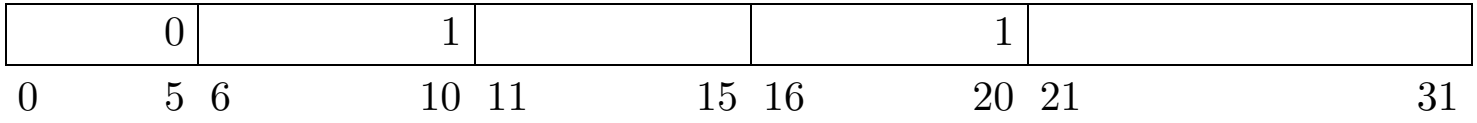
```
add r1, r2, r3 !  Add integer registers.
```



```
addf f1, f2, f3 !  Add FP registers, single precision.
```



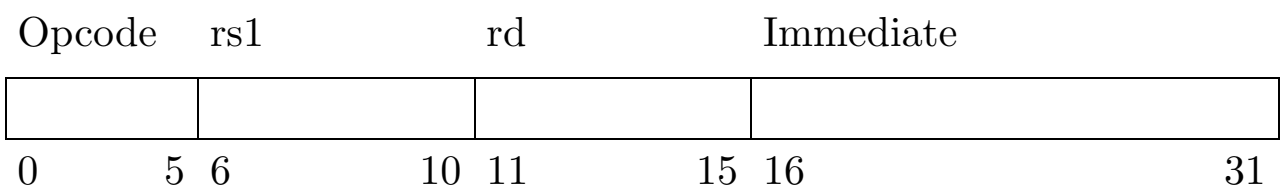
```
movi2s f1, r1 !  Move r1 to f1.  (rs2 operand not used.)
```



EE 4720 Lecture Transparency. Formatted 11:41, 25 August 2000 from lsli05.
05-12

05-13

Type I:



Used for loads, stores, some CTIs, and ALU immediate instructions.

Examples

```
addi r1, r2, #3     ! r1 = r2 + 3
lw   r2, 10(r3)     ! r2 = MEM[r3+10]
beqz r1, 20         ! if( r1 == 0 ) goto PC + 4 + 20 * 4  (rd unused).
jr   r1             ! goto r1
jalr r1             ! r31 = pc + 4; goto r1
```


EE 4720 Lecture Transparency. Formatted 11:41, 25 August 2000 from lsli05.
05-13

05-14

Type I Examples

```
addi r1, r2, 3 !  r1 = r2 + 3
```

addi	r2	r1	3
1	2	1	3
0 5	6 10	11 15	16 31

```
lw r2, 10(r3) !  r2 = MEM[r3+10]
```

lw	r3	r2	10
0x28	3	2	0xa
0 5	6 10	11 15	16 31

```
beqz r1, 20 !  if( r1 == 0 ) goto PC + 4 + 20 * 4 (rd unused).
```

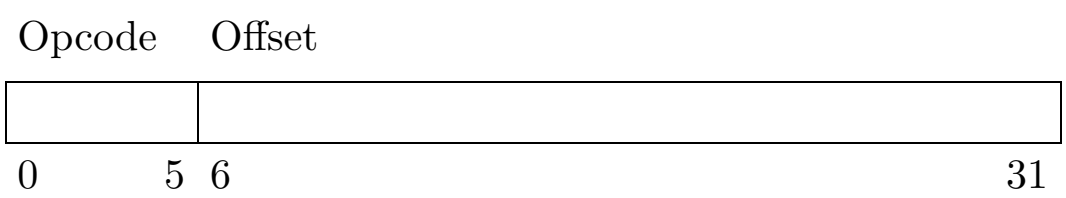
beqz	r1	Unused	20
0x1e	1		0x14
0 5	6 10	11 15	16 31

EE 4720 Lecture Transparency. Formatted 11:41, 25 August 2000 from lsli05.

05-14

05-15

Type J:

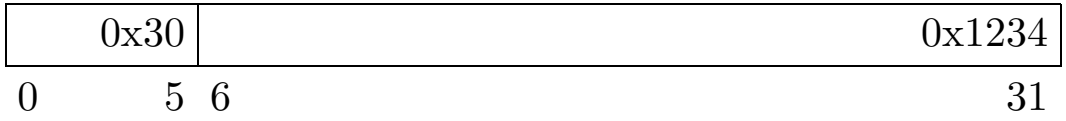


Used for jump and jump & link.

Examples:

```
j 0x1234            ! goto PC + 4 + 0x1234 * 4
jal 0x1234          ! r31 = PC + 4; goto PC + 4 + 0x1234 * 4
```

```
j 0x1234 !  goto PC + 4 + 0x1234 * 4
 j       0x1234
```



EE 4720 Lecture Transparency. Formatted 11:41, 25 August 2000 from lsli05.
05-15

05-16

Synthetic Instructions and DLX (NIB)

Misleading (in a nice way) assembly language mnemonics.

Implies a "new" opcode, but really uses an existing one.

Meant for programmer convenience.

Example, set register to zero:

```
clr r1              ! Synthetic instruction
add r1, r0, r0      ! True instruction (DLX)
```

Assembler generates a `add r1, r0, r0` when it finds a `clr r1` mnemonic.

Assembler "sees" synthetic instruction *but processor only sees true instructions.*

Sample Synthetic Instructions (NIB)

05-17

No Operation:

```
nop               ! Synthetic
add  r0, r0, r0   ! DLX
bnez r0, 0        ! DLX
```

Register move:

```
movi2i r1, r2     ! Synthetic
add    r1, r2, r0 ! DLX
```

Bitwise invert:

```
not  r1, r2       ! Synthetic
xori r1, r2,  #-1 ! DLX
```