RESUMEN DEL PROCESADOR DLX

El DLX es un procesador de tipo RISC académico, aunque inspirado en los primeros RISC y en particular en el MIPS (http://www.mips.com). Contiene 32 registros de 32 bits y de propósito general de R0 a R31 (General Purpose Registers, GPR), es decir todos sirven para todo. Las únicas excepciones son el registro R0 que siempre contiene el valor cero, y el R31, que por otras razones que se estudiarán más adelante, no se debe usar. Existen otros registros especiales y de coma flotante (FP) que no se estudian en este curso.
Todas las instrucciones ocupan una palabra de 32 bits exactamente, y deben estar alineadas en memoria (ocupar direcciones múltiplo de 4). O sea una instrucción no puede empezar en la dirección 20003 y acabar en la 20006; debe ocupar de la 20000 a la 20003 o bien de la 20004 a la 20007.

JUEGO DE INSTRUCCIONES. En el anexo 1 se tiene una tabla resumen de las instrucciones del DLX y una breve explicación del juego de instrucciones, del cual a continuación explicamos los detalles más importantes.

Transferencia de datos (Ld/St). Son muy simples ya que sólo pueden usar un modo de direccionamiento: Registro base más desplazamiento (puede usarse cualquier registro). Sólo se usa en las de transferencia de datos, y nunca en las ALU, las cuales sólo pueden usar registros o valores inmediatos como operandos. Por ejemplo, si usamos el paréntesis como notación del direccionamiento y el corchete como dirección de memoria, una instrucción de carga sería: LW R1, d16(R3) ; R1<— [R3+d16], donde R3 es el registro base en esta instrucción y d16 una constante con signo de 16 bits. Evidentemente un tercer operando no tiene sentido en estas instrucciones de transferencia de datos. Un almacenamiento sería: SW d16(R4), R5 ; [R4+d16] <— R5. Dado que el registro R0 contiene siempre el valor 0, si se usa R0 como registro base se está accediendo a los primeros 32Kb de la memoria o a los últimos 32KB (si el desplazamiento fuera negativo).
Existen variantes en función de si se lee (Load) un dato con o sin signo, y de si se lee/escribe una palabra de 32 bits (Word), una media palabra de 16 bits (Half word) o un Byte. Por ejemplo LHU lee de memoria 16 bits sin signo (Unsigned Half): LHU R7, -23(R0) ; R7 <— (extensión a 32 bits sin signo de la media palabra de [R0-23]) (téngase en cuenta que R0 siempre contiene 0).
Una característica habitual en todos los RISC, y que también se tiene en DLX, es la ausencia de la instrucción genérica de movimiento de datos MOV. Dada la simplicidad del acceso a memoria de éstos, no se permite ningún tipo de movimiento entre datos de memoria (habría que suplirlo con varios Ld/St), pero tampoco existe el MOV entre registros, ni tampoco de un valor inmediato a un registro. Estas últimas instrucciones se sustituyen por cualquier operación que no afecte al registro fuente, por ejemplo:

MOV R3, R4 se convierte en ADD R3, R4, R0 ; ya que R0 siempre vale 0.
O bien MOV R3, R4 se convierte en ADDI R3, R4, 0.
MOV R3, d16 se convierte en ADDI R3, R0, d16 ; ya que R0 siempre vale 0.

Aunque parece que esto va a llevar a una pérdida de tiempo, por tener que realizar una operación de suma donde no haría falta hacerla, se verá cuando se estudie la arquitectura del DLX que no implica ningún retraso o pérdida de rendimiento. Tan sólo se ha requerido la adición de una instrucción para cargar valor inmediato de 16 bits sobre la parte alta (16 bits MSB) de un registro de 32 bits. Por ejemplo, MOV R9, d32 no existe por razones de formato de instrucciones (una constante de 32 bis no cabe en una instrucción de 32 bits), y debe convertirse en:

LHI R9, d16 (altos)
ADDUI R9, R0, d16 (bajos)

Un último comentario: no existen instrucciones específicas para pila (tipo PUSH y POP). Se apuesta en su lugar por un uso masivo de registros, aunque siempre cabe la posibilidad de almacenar en memoria con load/stores e incrementar/decrementar un registro que actúe de puntero a memoria.

Instrucciones ALU. Como se ha visto en los ejemplos anteriores, son de tres operandos y además son todas registro-registro, es decir todo operando es un registro, y ningún operando puede estar en memoria. También se incluyen instrucciones con direccionamiento inmediato: Una constante de 16 bits actúa como segundo operando fuente. Por ejemplo: ALUoper R1,R2,#d16, ; R1 <— R2 *oper* d16 ; R2 y d16 son fuente, R1 destino. d16 es una constante de 16 bits. Sólo se puede usar en instrucciones ALU, y no en las de transferencia de datos. La almohadilla # puede suprimirse.
Nótese la regularidad de las instrucciones ALU: toda operación simple tiene otras tres variantes (siempre que lo permita). Por ejemplo para la suma con signo ADD, existe:

Suma sin signo ADDU.
Suma con signo y con direccionamiento inmediato (valor de 16 bits) ADDI
Suma sin signo y con direccionamiento inmediato (valor de 16 bits) ADDUI

Evidentemente en una operación lógica (OR, XOR, AND) no tiene sentido el signo. En una multiplicación o división por razones de implementación no existe el direccionamiento inmediato.
Finalmente existen unas instrucciones que valoran o inicializan a cierto o falso (en inglés "set") un registro destino comparando los dos fuente (sustituyen a las típicas instrucciones de comparación de procesadores como los x86). Están pensadas para instrucciones condicionales, por ejemplo SET if GREATER OR EQUAL (Inicializa comparando si es mayor o igual): SGE R13, R21, R22 ; R13 <—1 if (R21 >= R22). Existen las siguientes:

LT, if LESS THAN, si menor que
GT, if GREATER THAN, si mayor que
LE, if LESS OR EQUAL, si menor o igual que
GE, if GREATER OR EQUAL, si mayor o igual que
EQ, if EQUAL, si es igual que
NE, if NOT EQUAL, si no es igual que

Instrucciones de control de flujo Los saltos incondicionales (*jumps*) son relativos al PC (offset de 26 bits con signo) , o bien indican el nuevo PC en un registro: ej.:

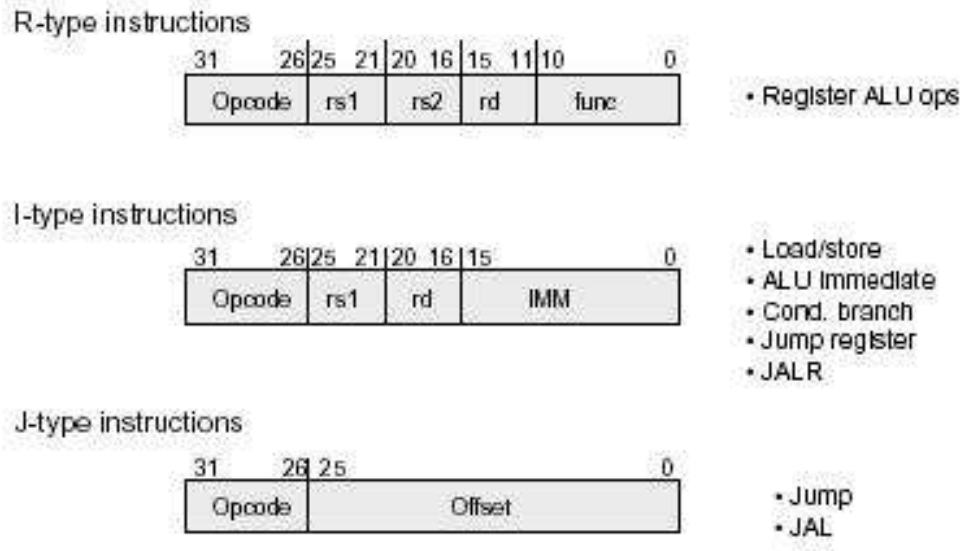
J etiqueta
JR R1

Para llamadas a funciones se usa JAL (*Jump and Link*) que salva la dirección de retorno en el registro R31 (reservado para esta función). Así, el retorno de función no requiere una instrucción expresamente dedicada como RET, sino que se usa JR R31. En caso de llamadas anidadas, basta con salvar R31 antes en algún sitio seguro (memoria o registro) y recuperarlo posteriormente.
Los saltos condicionales (*branch*) tienen dos campos: el registro que contiene la condición (1 ó 0) y el offset del PC (16 bits con signo). Así, la secuencia de comparación en un 8086 y en un DLX quedaría por ejemplo:

8086	DLX
CMP AX,BX ; *afecta al reg. de banderas PSW* JNE destino; *Si distintos salta a destino*	SNE R1, R2, R3 *; Compara R2,R3. Si distintos, R1=1.* BNEZ R1, destino ; *si R1=1, salta a destino*

Nótese cómo la relación (dependencia) entre la comparación y el branch es ahora explícita (R1), y no implícita (a través de un registro "oculto" como el PSW).

Finalmente, el formato de las instrucciones es muy regular y simple, como se muestra en la figura:



ANEXO 1: JUEGO DE INSTRUCCIONES DEL DLX

Instrucciones para la transferencia de datos	
LB *Rd,Adr*	Load byte (sign extension)
LBU *Rd,Adr*	Load byte (unsigned)
LH *Rd,Adr*	Load halfword (sign extension)
LHU *Rd,Adr*	Load halfword (unsigned)
LW *Rd,Adr*	Load word
LF *Fd,Adr*	Load single-precision Floating point
LD *Dd,Adr*	Load double-precision Floating point
SB *Adr,Rs*	Store byte
SH *Adr,Rs*	Store halfword
SW *Adr,Rs*	Store word
SF *Adr,Fs*	Store single-precision Floating point
SD *Adr,Fs*	Store double-precision Floating point
MOVI2FP *Fd,Rs*	Move 32 bits from integer registers to FP registers
MOVI2FP *Rd,Fs*	Move 32 bits from FP registers to integer registers
MOVF *Fd,Fs*	Copy one Floating point register to another register
MOVD *Dd,Ds*	Copy a double-precision pair to another pair
MOVI2S *SR,Rs*	Copy a register to a special register (not implemented!)
MOVS2I *Rs,SR*	Copy a special register to a GPR (not implemented!)
Instrucciones lógicas y aritméticas para enteros	
ADD *Rd,Ra,Rb*	Add
ADDI *Rd,Ra,Imm*	Add immediate (all immediates are 16 bits)
ADDU *Rd,Ra,Rb*	Add unsigned
ADDUI *Rd,Ra,Imm*	Add unsigned immediate
SUB *Rd,Ra,Rb*	Subtract
SUBI *Rd,Ra,Imm*	Subtract immediate
SUBU *Rd,Ra,Rb*	Subtract unsigned
SUBUI *Rd,Ra,Imm*	Subtract unsigned immediate
MULT *Rd,Ra,Rb*	Multiply signed
MULTU *Rd,Ra,Rb*	Multiply unsigned
DIV *Rd,Ra,Rb*	Divide signed
DIVU *Rd,Ra,Rb*	Divide unsigned
AND *Rd,Ra,Rb*	And
ANDI *Rd,Ra,Imm*	And immediate
OR *Rd,Ra,Rb*	Or
ORI *Rd,Ra,Imm*	Or immediate
XOR *Rd,Ra,Rb*	Xor
XORI *Rd,Ra,Imm*	Xor immediate
LHI *Rd,Imm*	Load high immediate - loads upper half of register with immediate
SLL *Rd,Rs,Rc*	Shift left logical
SRL *Rd,Rs,Rc*	Shift right logical
SRA *Rd,Rs,Rc*	Shift right arithmetic
SLLI *Rd,Rs,Imm*	Shift left logical 'immediate' bits
SRLI *Rd,Rs,Imm*	Shift right logical 'immediate' bits
SRAI *Rd,Rs,Imm*	Shift right arithmetic 'immediate' bits
S__ *Rd,Ra,Rb*	Set conditional: "__" may be EQ, NE, LT, GT, LE or GE
S__I *Rd,Ra,Imm*	Set conditional immediate: "__" may be EQ, NE, LT, GT, LE or GE
S__U *Rd,Ra,Rb*	Set conditional unsigned: "__" may be EQ, NE, LT, GT, LE or GE
S__UI *Rd,Ra,Imm*	Set conditional unsigned immediate: "__" may be EQ, NE, LT, GT, LE or GE
NOP	No operation

Instrucciones de Control	
BEQZ *Rt,Dest*	Branch if GPR equal to zero; 16-bit offset from PC
BNEZ *Rt,Dest*	Branch if GPR not equal to zero; 16-bit offset from PC
BFPT *Dest*	Test comparison bit in the FP status register (true) and branch; 16-bit offset from PC
BFPF *Dest*	Test comparison bit in the FP status register (false) and branch; 16-bit offset from PC
J *Dest*	Jump: 26-bit offset from PC
JR *Rx*	Jump: target in register
JAL *Dest*	Jump and link: save PC+4 to R31; target is PC-relative
JALR *Rx*	Jump and link: save PC+4 to R31; target is a register
TRAP *Imm*	Transfer to operating system at a vectored address; see Traps.
RFE *Dest*	Return to user code from an exception; restore user mode (not implemented!)

Instrucciones en punto flotante	
ADDD *Dd,Da,Db*	Add double-precision numbers
ADDF *Fd,Fa,Fb*	Add single-precision numbers
SUBD *Dd,Da,Db*	Subtract double-precision numbers
SUBF *Fd,Fa,Fb*	Subtract single-precision numbers.
MULTD *Dd,Da,Db*	Multiply double-precision Floating point numbers
MULTF *Fd,Fa,Fb*	Multiply single-precision Floating point numbers
DIVD *Dd,Da,Db*	Divide double-precision Floating point numbers
DIVF *Fd,Fa,Fb*	Divide single-precision Floating point numbers
CVTF2D *Dd,Fs*	Converts from type single-precision to type double-precision
CVTD2F *Fd,Ds*	Converts from type double-precision to type single-precision
CVTF2I *Fd,Fs*	Converts from type single-precision to type integer
CVTI2F *Fd,Fs*	Converts from type integer to type single-precision
CVTD2I *Fd,Ds*	Converts from type double-precision to type integer
CVTI2D *Dd,Fs*	Converts from type integer to type double-precision
__D *Da,Db*	Double-precision compares: "__" may be EQ, NE, LT, GT, LE or GE; sets comparison bit in FP status register
__F *Fa,Fb*	Single-precision compares: "__" may be EQ, NE, LT, GT, LE or GE; sets comparison bit in FP status register

Directivas del simulador WinDLX	
.align *n*	Cause the next data/code loaded to be at the next higher address with the lower n bits zeroed (the next closest address greater than or equal to the current address that is a multiple of 2n (e.g. .align 2 means next word begin).
.ascii *"string1","..."*	Store the "strings" listed on the line in memory as a list of characters. The strings are not terminated by a 0 byte.
.asciiz *"string1","..."*	Similar to .ascii, except each string is terminated by a 0 byte.
.byte *byte1,byte2,...*	Store the bytes listed on the line sequentially in memory.
.data *[address]*	Cause the following code and data to be stored in the data area. If an address was supplied, the data will be loaded starting at that address, otherwise, the last value for the data pointer will be used. If we were just reading data based on the text (code) pointer, store that address so that we can continue from there later (on a .text directive).
.double number1,...	Store the "numbers" listed on the line sequentially in memory as double-precision Floating point numbers.
.global *label*	Make label available for reference by code found in files loaded after this file.
.space *size*	Move the current storage pointer forward size bytes (to leave some empty space in memory)
.text *[address]*	Cause the following code and data to be stored in the text (code) area. If an address was supplied, the data will be loaded starting at that address, otherwise, the last value for the text pointer will be used. If we were just reading data based on the data pointer, store that address so that we can continue from there later (on a .data directive).
.word *word1,word2,...*	Store the word listed on the line sequentially in memory.