Memória Técnica

sobre l'etapa inicial del processador de l'assignatura PEC

1. Instruccions Implementades:

Tipus d'instrucció	Implementada	Funcional
Aritmetico-lògiques	**Y**	**Y**
Comparacions	**Y**	**Y**
Moves	**Y**	**Y**
Immediates	**Y**	**Y**
Load-Store	**Y**	**Y**
Multiplicació	**Y**	**Y**
Divisió	**Y**	**Y**
Salts Condicionals	**Y**	**Y**
Salts Incondicionals	**Y**	**Y**
Especials(*HALT*)	**Y**	**Y**

2. Controlador de Memoria:

Vàrem decidir implementar el controlador de memòria sense estats, només basant-nos en els flancs de rellotge. La memòria és tan ràpida comparada amb el nostre processador que no ens cal anar amb cura sobre els *timings* ni tenir múltiples estats per assegurar-nos de respectar els cronogrames de lectures o escriptures. De fet, seria possible fins i tot treure el divisor de freqüència del rellotge de 50 MHz i aquest controlador hauria de continuar funcionant, tot i que no tenim en consideració els retards dels diferents components, que en una arquitectura funcional sí s'haurien de tenir en compte.

Joan Teruel; Albert Azcárate

3. Decisions de Disseny:

I. Unitat de control:

Per poder diferenciar entre instruccions reals i instruccions no implementades o invalides hem fet ús de dos senyals (*ir_interna* i *op_code_ir_pre*). La primera es el valor del *Instruction Register* filtrat. En el cas que es detecti que qualsevol bit d'entrada es diferent a 1 o 0, *IR* passarà a valdre *x"FFFE"* (implementada com un *NOP*) per no introduir valors erronis al computador.

Per altre banda, *op_code_ir_pre* es el codi d'operacio previ a comprobar si la instrucció es ilegal o no implementada (per exemple una instruccio DIV(OpCode = 8) amb codi de funció no implementat (FCode = 6)). Si es detecta que es ilegal la instruccio passarà a ser un *NOP*.

II. Unitat de Multicicle:

La imposició d'un estat *Fetch* i un *Decode* ens ha portat a una màquina d'estats binària i pocs vèrtexs. Sempre passem de *Fetch* a *Decode* i viceversa. En general sempre passem al *datapath* les senyals de control en estat *Decode* i les bloquegem en estat *Fetch*.

III. Unitat Aritmètico-Lògica:

Hem decidit implementar la ALU de manera unitària. No té mòduls interns, sinó que està implementat tot directament, multiplexant les sortides segons l'Operació i la Funció d'Operació. Aquesta decisió no és gaire crítica i fàcilment es podrien fer mòduls per fer certes operacions com SHL o SHA sense generar tant creuament de llibreries i complicació de codi.

IV. Controladors de memòria i SRAM:

Hem escollit l'esquema de SRAM de *words* de 16 bits i 32k files, ja que es la més òptima segons l'esquema *Fecth-Decode*. Com ja s'ha especificat abans, no tenim dificultats en complir els cronogrames de la SRAM, així que la implementació és bastant directa, actualitzant entrades i sortides segons flancs ascendents de rellotge.

V. Instruccions de Salts:

Per instruccions de control de flux hem decidit ampliar el *ldpc* a un *std_logic_vector* de tamany 2, per distingir quan hem de fer Pc = Pc + 2, Pc = RegN, Pc = Pc + 2 + Imm8, Pc = Pc. Hem augmentat el multiplexor d'entrada del Banc de Registres perquè tingui entrades des de Memòria, ALU i el Pc.

VI. Altres aspectes:

Hem afegit una serie de condicions perquè el processador pugui discriminar entre operacions 'legals' i aquelles que encara no estan implementades, que es comporten en la práctica com a *NOP*.

D'acord amb les nostres consideracions, hem detectat que no està especificat al PDF de la implementació de l'*ALU* si es permet o no la divisió entre zero; hem fet que es detecti si es divideix per zero i retorni *x"XXXX"* com a resultat.

Tampoc hem vist que estigues especificat que fer si un *Jump* o *Branch* salta a la zona de dades (@ < *x"C000"*) o si es fa overflow del PC. En el primer cas hem decidit no saltar i fer PC = PC + 2, com si fos un salt no agafat. En el segon cas si arribem a PC = x"FFFE", no actualitzarem mai el pc i ens quedarem repetirn la mateixa instruccio indefinidament.

4. Comentaris:

Hem comentat múltiples seccions del nostre codi VHDL per clarificar les accions que les diferents instruccions executen, també per ajudar-nos a debugar al modelsim, hem creat una sèrie de classes com 'op_code' on emmagatzemem les diferents nomenclatures que tenen les instruccions SISA que conjuntament amb una sèrie de comandes i senyals ens ajuda a veure els valors dels cronogrames que en el modelsim es mostren. Si en algun moment es vol consultar el *RTL Viewer* de Control_l recomanem comentar dins del codi Control_l.vhd des de l'etiqueta "-- MODELSIM SIGNALS" fins a "-- END MODELSIM SIGNALS", ja que això redueix significativament els components fets servir al *RTL Viewer*, perquè aquesta secció l'utilitzem per forçar al 'wave' a mostrar els valors amb les nomenclatures desitjades.