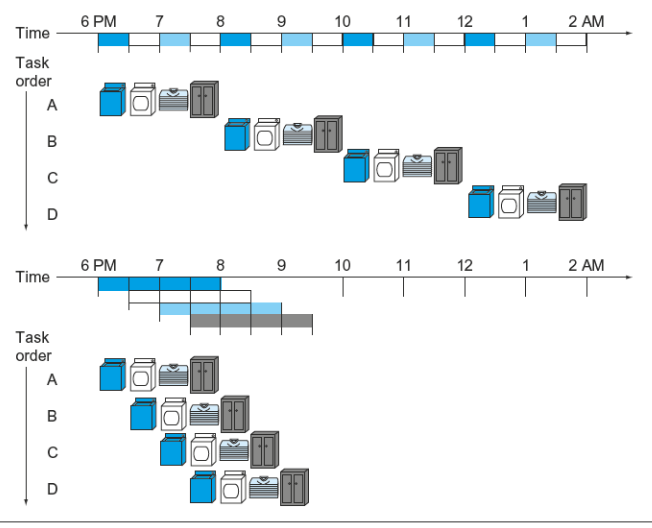
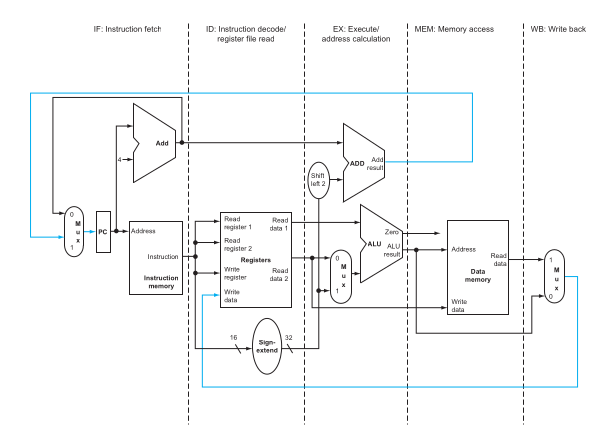
Bueno, el tp final de arquitectura, hace un año que lo tenemos para hacer, pero el pija de braian me soltó la mano como siempre y no ayuda en nada.

Lo primero que tenemos q entender es que un pipeline es la forma en la que se diseña un procesador, precisamente los procesadores RISC, que son procesadores con un set reducido de instrucciones. La idea es llevar cada instrucción a su mínima expresión para utilizar la menor cantidad de hardware posible y así poder simplificar la implementación. por q es necesario esto? porq los procesadores complejos se encontraron con el problema de que la velocidad de procesamiento, osea, la frecuencia a la q funcionan, estaba limitada, no solo por cuestiones fisicas, de temperatura, sino por el camino q esteban me va a decir por wsp… camino critico! que define una frecuencia maxima. Entonces, si separamos toda la ejecucion de una instruccion en varias etapas, ese camino se divide y por lo tanto podemos aumentar la frecuencia del clock. Aunque por otro lado vamos a necesitar realizar más operaciones, para poder reemplazar las operaciones más complejas, y también vamos a necesitar duplicar cierto hardware para que cada etapa sea independiente de las demas. Por lo pronto, con esta definicion nos quedamos tranquilos.

Trtatando de hacer una analogia, el libro nos lleva a comparar la ejecucion de una instruccion con el ciclo de de limpieza de ropa, donde la ropa va pasando por diferentes etapas desde q esta sucia hasta que esta limpia:  
  


Como podemos ver, si separamos el lavado en diferentes etapas, podemos reutilizar cada etapa una vez que la tanda de ropa (instruccion) pasa a la siguiente estacion; por ende, luego de 4 ciclos tendriamos una instruccion finalizada, y ademas, de ahi en más, por cada ciclo tendriamos una nueva instruccion finalizada. Todo esto teniendo en cuenta que cada etapa es independiente de las demas. Luego de todo este analisis, surgen diferentes problemas, como por ejemplo, que sucederia si la instruccion a ejecutar es un jump. Bueno, en ese caso recien podriamos saberlo cuando la instruccion es decodificada en la etapa InstructionDecode, dnd ya habriamos metido otra instruccion en la etapa InstructionFetch. Estos problemas se solucionan agregando unidades de control y otras unidades que analizaremos cuando sea necesario.

Empezamos con una implementacion simple, sin tener en cuenta los problemas derivados de utilizar pipelining. Arrancamos cargando en la memoria todas las instrucciones sin la utilizacion de UART y por ende, tampoco la unidad de debugging (con la que frenamos el pipeline para ir viendo los resultados parciales y cargamos los datos en memoria). Tampoco la unidad de control ni la forwarding unit. No hay control de hazards (peligros!).  
  
Osea, siguiendo la implementacion del libro “Computer organization and design”.  
  
Pipeline simple:  
  


Podemos ver que el pipeline es de 5 etapas:

IF (Instruction Fetch): donde se saca la instrucción de la memoria de instrucciones usando el program counter.

ID (Instruction Decode): dnd se decodifica la instruccion para saber de que tipo es.

EX (execute): dnd se ejecuta la instruccion utilizando la ALU.

MEM (Memory access): se accede a la memoria de datos para buscar los operandos.

WB (Write Back): donde se escribe el resultado obtenido en los registros internos.

Hay diferentes tipos de instrucciones:  
  
R-type

SLL, SRL, SRA, SLLV, SRLV, SRAV, ADDU,

SUBU, AND, OR, XOR, NOR, SLT

I-Type

LB, LH, LW, LWU, LBU, LHU, SB, SH, SW

ADDI, ANDI, ORI, XORI, LUI

SLTI, BEQ, BNE, J, JAL

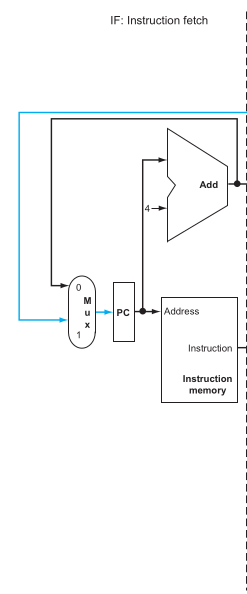
J-Type

JR, JALR

Que no tengo bien claro como se definen, pero hice una tabla con todos los opcodes y en el otro archivo “Instrucciones MIPS” se explican en detalle.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| R-type | OpCode (5-0) | I-type | OpCode (31-26) | J-type | OpCode (5-0) |
| SLL | [000000] | LB | [100000] | JR | [001000] |
| SRL | [000010] | LH | [100001] | JALR | [001001] |
| SRA | [000011] | LW | [100011] |  |  |
| SLLV | [000100] | LWU | [100111] |  |  |
| SRLV | [000110] | LBU | [100100] |  |  |
| SRAV | [000111] | LHU | [100101] |  |  |
| ADDU | [100001] | SB | [101000] |  |  |
| SUBU | [100011] | SH | [101011] |  |  |
| AND | [100100] | SW | [101011] |  |  |
| OR | [100101] | ADDI | [001000] |  |  |
| XOR | [100110] | ANDI | [001100] |  |  |
| NOR | [100111] | ORI | [001101] |  |  |
| SLT | [101010] | XORI | [001110] |  |  |
|  |  | LUI | [001111] |  |  |
|  |  | SLTI | [001010] |  |  |
|  |  | BEQ | [000100] |  |  |
|  |  | BNE | [000101] |  |  |
|  |  | J | [000010] |  |  |
|  |  | JAL | [000011] |  |  |

Arrancando entonces, nos concentramos en la 1era etapa, Instruction Fetch:

Bueno, la poronga esta tiene un multiplexor, que

define si la entrada al PC viene de la alu (un sumador simple en realidad) o de la etapa de ejecucion, supongo será cuando hacemos un jump o alguna otra instruccion condicional; lo vamos a dejar, pero el cable q viene de EX lo vamos a dejar al aire, así nomas, para q funcione para el orto el pipeline. El registro de PC de cuentos bits es? depende del tamaño de la memoria obviamente, en el BIP era de 10, pero ahora anda saber. le ponemos 32 y despues usamos los q nos pinte.

La memoria la sacamos del BIP, es un modulo q ya viene en el VIvado, un template. Y bueno, para el sumador, usamos una ALU q le hardcodeamos la operacion en SUMA y listo.

Tampoco es la locura la etapa asiq vamos con el codigo en verilog.

`timescale 1ns / 1ps

module InstructionFetch(

input clk, reset,

output pc,

output [31:0] instruccion

);

wire BIP\_enable, finish\_program, WrRAM, RdRAM, clk, locked;

wire[10:0] PC, Addr;

wire[15:0] Program\_Data, In\_Data, Out\_Data;

wire[31:0] out\_Acc\_Counter;

UART uart(rx, clk, reset, finish\_program, out\_Acc\_Counter, tx, BIP\_enable);

Data\_memory #(.INIT\_FILE("/home/vlad/Arquitectura2018/datos.txt")) Data\_memory(.Wr(WrRAM),.Rd(RdRAM),.clk(clk),.ena(BIP\_enable), .Addr(Addr), .In\_Data(In\_Data), .Out\_Data(Out\_Data));

Program\_memory #(.INIT\_FILE("/home/vlad/Arquitectura2018/instrucciones.txt")) Program\_memory (0, clk, BIP\_enable, PC, 0, Program\_Data);

CPU bip(BIP\_enable, clk, reset, Program\_Data, Out\_Data, In\_Data, PC, WrRAM, RdRAM, finish\_program);

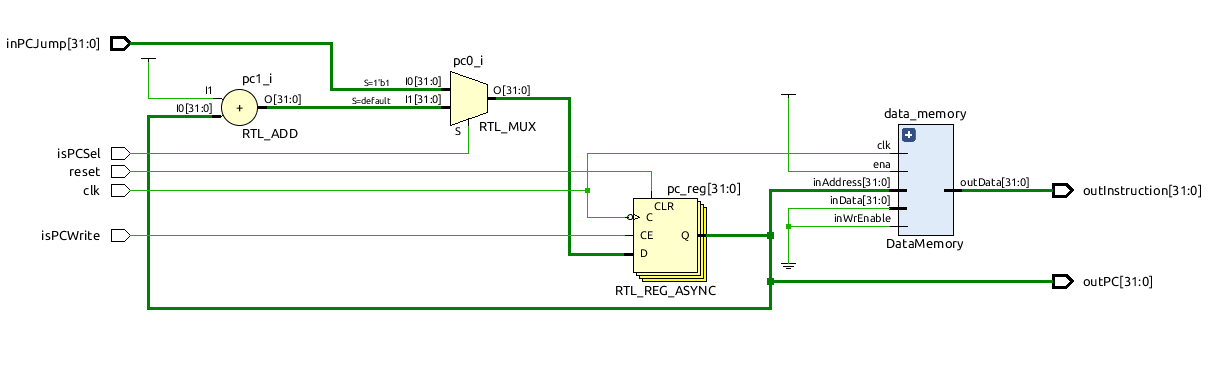
assign out\_Acc\_Counter[31:27] = 0;

assign out\_Acc\_Counter[26:16] = PC;

assign out\_Acc\_Counter[15:0] = In\_Data;

assign Addr = Program\_Data[10:0];

endmodule

InstructionFetch diagram