4.4 A Simple Implementation Scheme

Locations of Fields within Machine Code Instructions:

opcode Always exists and always located at bits 31:26 (6 bits)

If the following fields exist in a particular instruction they are always found at

rs bits 25:21 (5 bits)

rt bits 20:16 (5 bits)

rd bits 15:11 (5 bits)

shamt bits 10:6 (5 bits)

funct bits 5:0 (6 bits)

imm-16 bits 15:0 (16 bits)

referred to as immediate or address depending on use

imm-26 bits 25:0 (26 bits)

General rule for fast hardware:

Assume the field exists and use it

Ignore the garbage result of using the field through control values if it does not exist

Control of the Arithmetic and Logic Unit (ALU):

Use combinational logic to determine if ***opcode*** (bits 31:26)

1) is not 000000

4-bit ALU operation can only be

a) add for memory address calculation (**`lw`**, **`sw`**), or

b) sub for comparison (**`beq`**)

c) ALU not used so, 4-bit ALU operation is a don't care

2) is 000000

4-bit ALU operation depends on ***funct*** (bits 5:0, R-type instruction)

All Other Control Signals (1 bit each, see Figure 4.15)*:*

Use combinational logic determine other control signals using ***opcode*** (bits 31:26)

RegDst — Is ***rt*** or ***rd*** the register to write to (a don't care if no register written)?

Note that load word writes to ***rt***, all other instructions use ***rt*** as a source register

RegWrite — Is any register to be written to?

If output of ***rt***/***rd*** MUX is garbage because neither field is a destination register, we don't want to trash the value in some random register

ALUSrc — Does the ALU take R[***rt***] or ***SignExtImm*** as its second input (if any)?

If ***opcode*** = 000000, then instruction is R-type, so deassert ALUSrc and use register file output, otherwise use output of sign extension unit (I-type)

If instruction is neither R-type nor I-type ALUSrc is a don't care since the ALU output will not be used

MemWrite — Does the Data Memory Unit write anything to memory?

Don't want to write to memory unless instruction is a store

MemRead — Does the Data Memory Unit read anything from memory?

Don't want to read from memory unless instruction is a load

MemtoReg — Is Data Memory Unit output or is ALU output sent to the register file?

If instruction is a load, send Data Memory Unit output to Register File (assert MemtoReg), if instruction is R-type send ALU output to Register File (deassert MemtoReg)

If instruction is neither load nor R-type, MemtoReg is a don't care since "Write Data" on the Register File is garbage either way and deasserted RegWrite will block its use

PCSrc — Is PC+4 or is PC+4+BranchDistance sent to the PC?

In Chapter PCScr is formed by ANDing Branch and the Zero output of the ALU

If **`bne`** instruction also existed, calculation of PCScr is slightly more complicated

Adding Jump Capability (see Figure 4.24)*:*

Add MUX to select jump address or PC+4/Branch address

Use combinational logic determine if ***opcode*** (bits 31:26) is

1) 000010

Instruction is a jump (**`j`**), assert Jump control signal

2) not 000010

Instruction is not a jump, deassert Jump control signal

Note: the only J-type instruction in Chapter 4 is jump. Full implementation would need to also handle jump-and-link (jal) and jump register (jr)

4.5 Overview of Pipelining (partial - to be continued next time)

Operations Performed by Various Types of Instructions:

Instruction Fetch (IF)

All instructions - get the machine code instruction from instruction memory

Instruction Decode (ID), Register Read, Sign Extension

All instructions -

ID - set control signals based on ***opcode***/***funct*** fields of instruction

Read ***rs*** and ***rt*** registers (even if the rs and/or rt are garbage)

Create ***SignExtImm*** (even if this is garbage)

Note that always reading two registers allows parallel operation of all three

None can occur until machine code instruction is available

ALU Execution (EX)

All instructions - ignore the result if it is not used

Can not occur until control signals, register read and (possibly) ***SignExtImm*** are available

Memory Read or Write (MEM)

Read from data memory if instruction is a load

Write to data memory if instruction is a store

Do nothing if instruction is neither load nor store

Can not occur until data address is calculated by ALU

Register Write (WB)

Write value to register if instruction is a load or R-type

Otherwise do nothing

Can not occur until ALU or data memory value available

Dependencies:

Situation where the proper execution of an instruction depends on the result of a previous instruction being available

Only the instruction set definition needs to be considered to determine dependences

The *implementation* of the instruction set is irrelevant

(Pipeline) Hazards:

Situation where the implementation of an instruction set results in incorrect operation of a given sequence of instructions (possibly due to dependencies)

Hazards may be removed by adding extra *forwarding* (also call *bypass*) hardware to the implementation pipeline

Hazards not removed using additional hardware must be avoided in machine code programs:

Assemblers and compilers should alter machine code output to avoid all hazards

Instruction Set Architectures must specify which hazards are not allowed in the machine code

Types of Pipeline Hazards:

Structural Hazards:

Available hardware units (memories, register files, ALUs, etc.) not sufficient to do all operations currently in pipeline

Don't exist in Classic 5-Stage MIPS pipeline unless the memory system does not allow simultaneous access to Instruction Memory and Data Memory

Data Hazards:

Situation where one instruction in pipeline has not yet made its result available to a subsequent instruction that needs it

Control Hazards:

Situation when fetch of the proper next instruction cannot occur because result of a branch or jump is not known (correct next PC value not yet available)