MICROELECTRONIC SYSTEMS DESIGN RESEARCH GROUP

FB Elektrotechnik und Informationstechnik
Prof. Dr.-Ing. Norbert Wehn
Dozent:
Uwe Wasenmüller
Raum 12-213, wa@eit.uni-kl.de

TECHNISCHE UNIVERSITÄT KAISERSLAUTERN

Task 5

Introduction

Subject of the this task is the extension of the fundamental pipelining concept, which was discussed in the previous task. The existing pipeline is to be expanded with a pipeline controller, which has the duty to resolve existing data dependencies. Situations in which the execution of the next instructions in the data path in its assigned cycle is prohibited are called *Hazards*.

Pipeline-Conflicts

There are three types of conflicts which have to be be distinguished:

1. *Structure hazards* occur if there is a resource conflict. This occurs, if the hardware cannot execute all possible combinations of instructions simultaneously.
2. *Data hazards* occur if instructions depend on the result of previous instructions caused by the overlapped execution (pipelining).
3. *Control hazards* result by scheduling instructions into the pipeline which are actually invalid because the program counter (PC) was modified (e.g. by a branch).

Structure Hazards

In the current implementation of the DLX no structure hazards can occur, because there are enough hardware resources for every combinations of instructions which are possible (e.g.. two read ports and a write port for register memory and separated data and instruction access.)

Data Hazards

The most common pipeline conflicts are of this type. Three subtypes can be distinguished according to the source of the hazard. In the following description of these subtypes we assume a sequence of instructions with *i* and *j*, with instruction *i* is issued directly before instruction *j* in the machine program. The occurring conflicts are named after there original order of execution:

- RAW (read after write) – *j* tries to read from a source before *i* can write it. So *j* faultily reads the old value.
- WAR (write after read) – j tries to write to a destination before it can be read by *i*, *so i* reads the new value.
- WAW (write after write) – *j* tries to write to an operand, before it is written by *i*. The write operations are executed in the wrong order and so the result of *i* remains in the destination instead of *j*.

In the existing DLX implementation only RAW data hazards can occur because operands are already read from the processor registers in the ID phase (early reading), and writing is exclusively performed in the transition from the MEM- to the WB-phase, so also no WAW hazards can occur.

The appearance of RAW-hazards is a result of the pipelining concept and the associated quasi simultaneous execution of sequential dependent instructions. The temporal difference from the read and write operations on the internal register memory, which happen in different pipeline stages, result in a delay of two cycles until the result of an instruction is written in a register. If one of two of the following instructions read operands from the destination register of the current instruction, they read the faulty old value because the result is not yet written.

Example:

```
ADD    R1, R2, R3
SUB    R4, R1, R5
SUB    R6, R1, R2
```

	cycle					
instruction	1	2	3	4	5	6
ADD	IF	ID	EX	MEM	WB	
SUB		IF	ID	EX	MEM	WB
SUB			IF	ID	EX	MEM

The ADD instruction writes the destination register R1 in the write back phase (WB). The subsequent SUB instructions require this value of register R1, to subtract it with R5 and R2 respectively. During the WB of ADD, the immediately following SUB instruction is in the MEM phase, the following instruction is in the EX phase. The operands are already read in the ID phase and therefore the wrong value is used in their subtractions. Subsequent instructions which use the value of R1 will produce the right result.

The described problem can generally be avoided by careful programming, resorting the instructions and placing independent instructions in between or inserting NOP instructions to delay the read operations after the write back. Disadvantage of this approach is the massive loss of throughput.

While the processing of the operand is done in the EX phase with all arithmetical and logical operations, an unresolved data dependency can fudge the branch behavior because register values can be loaded in the ID phase to calculate the jump address (see BEQZ, JR, ...). The same considerations as above apply.

Control Hazards

Control hazards occur if the program is not continued sequentially, but branches to another point in the program (jump instructions, subroutine calls, interrupt handlers, …).
In the existing implementation the calculation of the jump address is performed before the decode phase to detect branches as early as possible. In case of a branch the subsequent instruction is already fetched in the IF phase. This results in the subsequent instruction is performed regardless if the branch instruction is executed or not. This type of branch processing is called *Delayed Branch* and the subsequent instruction to the branch is called *Delay Slot*.

Forwarding

Forwarding represents an implementation concept which can resolve most (but not all!) data dependencies. A possible way to realize this on ALU operations works as follows: The ALU result pipeline register is always relayed to the ALU input and is therefore available earlier. When the Forwarding hardware detects that the instruction which is to be processed by the ALU has an operand from an register which is the destination of the previous instruction, the control hardware changes the source of the operand to the pipeline register which holds the result of the previous ALU operation.

The pipeline control unit which is to be designed must set various multiplexer control signals according to the ID-, EX-, and MEM-phase so that the correct values are loaded into the pipeline registers instead of the invalid register values. To detect such cases the opcodes and register addresses of all instruction combinations have to be compared. You must complete the VHDL source code file `dlx_pipe_ctrl.vhd`. For simplification the instructions are categorized in instruction type to reduce the number of possible combinations.

The instruction types in detail:

- NOFORW
  This type contains instructions which don't need forwarding because they don't access registers (i.e. J, JAL, TRAP, RFE, NOP and all instructions which have R0 as destination register).
- LOAD
  This type contains all load instructions (LB, LBU, LH, LHU, LW). The contents of the register file (rs1) is needed in the EX phase to calculate the address, but the to-be-written read date (rd) is not be used until the end of the MEM phase.
- STORE
  This type contains all store instructions (SB, SBU, SH, SHU, SW). The content of the register file (rs1) is needed in the EX phase to calculate the address, the date (rd) is needed in the MEM phase.
- RR_ALU
  This type contains all arithmetic operations with two register addresses (rs1 and rs2). The content of the register file is needed in the EX phase, the date (rd) to be written can be used from the ALU-result-pipeline-register for Forwarding.
- IM_ALU
  This type contains all arithmetic operations with one register address (rs1) and one direct operand. The content of the register file is needed in the EX phase, the date (rd) to be written can be used from the ALU-result-pipeline-register for Forwarding.
- BRANCH
  This type contains all branch instructions which read one register operand (rs1) (BEQZ, BNEZ, JR, JALR). The content of the register file is processed already in the ID phase.

As mentioned before not all data dependencies can be resolved by Forwarding. The data from a LOAD instruction for example are first available in the MEM phase. A immediately following ALU operation has to be stalled until the memory access was performed. Similar problems occur on branch instructions which need the contents of the register file in the ID phase, but the previous ALU operation may have not calculated the results yet.

The input signals `id_opcode_class, id_irl_rs1` and `id_ir_rs2` contain instruction type and source registers of the current instruction in the ID phase; `id_ex_opcode_class, id_ex_ir_rs1, id_ex_ir_rs2` and `id_ex_reg_rd` contain instruction type, source and destination register addresses of the current instruction in the EX phase. Analogous the signals `ex_mem_opcode_class, ex_mem_reg_rd` and `mem_wb_opcode_class, mem_wb_reg_rd` contain instruction type and destination of the MEM and WB phase.

These input signals are required to set five control signals. One signal controls a pipeline stall which halts the IF and ID phase to resolve data dependencies:

- stall
  This signal must be set to '1' if a data dependency is detected which cannot be resolved by Forwarding. There is a separate process (`stall_crtl`) in the module `dlx_pipe_ctrl` which only controls this signal.

The other signals control multiplexers at the inputs of the functional units, which can select a pipeline register instead of the content of the register file. These signals are controlled in the process `pipe_ctrl`:

- id_a_fwd_sel
  The ID phase may need a different value for BRANCH instructions than the one from the register file.
- ex_alu_opa_sel
  The input A of the ALU. Without Forwarding the content of the register address rs1 is put through.
- ex_alu_opb_sel
  The input B of the ALU. Without Forwarding the content of the register address rs2 or the direct operand is put through.
- ex_dm_data_sel
  The data input of the data memory. On store instructions the content of the register rd is put through.

The range of the values of these signals are FWDSEL_NOFORW (Forwarding not neccesary), FWDSEL_EX_MEM_ALU_OUT (result of the EX phase is applied to the input multiplexer) and FWDSEL_MEM_WB_DATA (the result in the MEM phase is applied to the input multiplexer).

For testing your implementation of the pipeline control unit with forwarding use the test bench of the DLX-pipeline as in task 4. The assembler test program is shown in the appendix. Corresponding machine code is given by `/asm/sieb_forw.out.` This test program realizes again a prime number calculation as in task 4. Functionality of the program is slightly extended compared to test program for task 4 and another assembler implementation was chosen to test the pipeline control unit.

Appendix – Sample Program for testing (prime number calculation)

```
main:      addui r1, r0, 1      ; r1: constant 1              00000000 24010001
           lw    r2, N(r0)      ; r2: N                       00000004 8c0200a0
           addui r3, r0, A      ; r3: base address of A       00000008 24030400

; initialization loop
; ========================
;  for( a[1]=0, i=2; i<= N; i++ )
;    a[i] = 1;

           addui r4, r0, 2      ; r4: i = 2                   0000000c 24040002
           sb    1(r3), r0      ; A[1] = 0                    00000010 a0600001
init_loop: addu  r5, r3, r4     ; r5 : address of A[i]        00000014 00642821
           sb    0(r5), r1      ; A[i] = 1                    00000018 a0a10000
           addui r4, r4, 1      ; i++                         0000001c 24840001
           sle   r6, r4, r2     ; r6: i <= N ?                00000020 0082302c
           bnez  r6, init_loop  ; if r6 == 1  loop init_loop  00000024 14c0ffec
           nop                  ;                             00000028 0000003f

; Main Loop
; =============
; for( i=2; i<=N/2; i++ )
;   for( j=i+i; j<=N; j+=i )
;     a[j] = 0;

            addui r4, r0, 2     ; r4: i = 2                   0000002c 24040002
            srai  r5, r2, 1     ; r5: N/2                     00000030 5c450001
outer:      slli  r6, r4, 1     ; r6; j = 2*i;                00000034 50860001
inner:      addu  r7, r3, r6    ; r7 : address of A[j]        00000038 00663821
            sb    0(r7), r0     ; A[j] = 0                    0000003c a0e00000
            addu  r6, r6, r4    ; j = j + i                   00000040 00c43021
            sle   r8, r6, r2    ; r8: j <= N ?                00000044 00c2402c
            bnez  r8, inner     ; if r8 == 1 loop inner       00000048 1500ffec
            nop                 ;                             0000004c 0000003f
            addui r4, r4, 1     ; i++                         00000050 24840001
            sle   r8, r4, r5    ; r8: i <= N/2 ?              00000054 0085402c
            bnez  r8, outer     ; if r8 == 1 loop outer       00000058 1500ffd8
            nop                 ;                             0000005c 0000003f

; Output of the prime numbers beginning at address 256
;=========================================

           addui    r4, r0, 2   ; r4: i = 2                   00000060 24040002
           addui    r7, r0, B   ; &B                          00000064 24070800

out_loop:  addu     r6, r3, r4  ; r6 = &A[i]                  00000068 00643021
           lbu      r8, 0(r6)   ; r8 = A[i]                   0000006c 90c80000
           beqz     r8, continue ;                            00000070 1100000c
           nop                  ;                             00000074 0000003f
           sw       0(r7), r4   ;                             00000078 ace40000
           addui    r7, r7, 4   ; r7 += 4                     0000007c 24e70004

continue:  addui    r4, r4, 1   ; r4 += 1                     00000080 24840001
           sle      r5, r4, r2  ; r5 = (i <= N)               00000084 0082282c
           bnez     r5, out_loop ;                            00000088 14a0ffdc

           nop                  ;                             0000008c 0000003f
           nop                  ;                             00000090 0000003f
           nop                  ;                             00000094 0000003f
           nop                  ;                             00000098 0000003f
           trap     0           ;                             0000009c 44000000

N:         .word    100         ;                             000000a0 00000064
           .align   10
A:         .space   101
           .align   10
B:         .space   101
```

Literature

[HEN/PAT96]	John L. Hennessy, David A. Patterson Computer Architecture - A Quantitative Approach, 2. Edition Morgan Kaufmann Publishers, Inc. San Francisco, California
[SAI/KAE]	Philip M. Sailer, David R. Kaeli The DLX Instruction Set Architecture Handbook Morgan Kaufmann Publishers, Inc., San Francisco, California