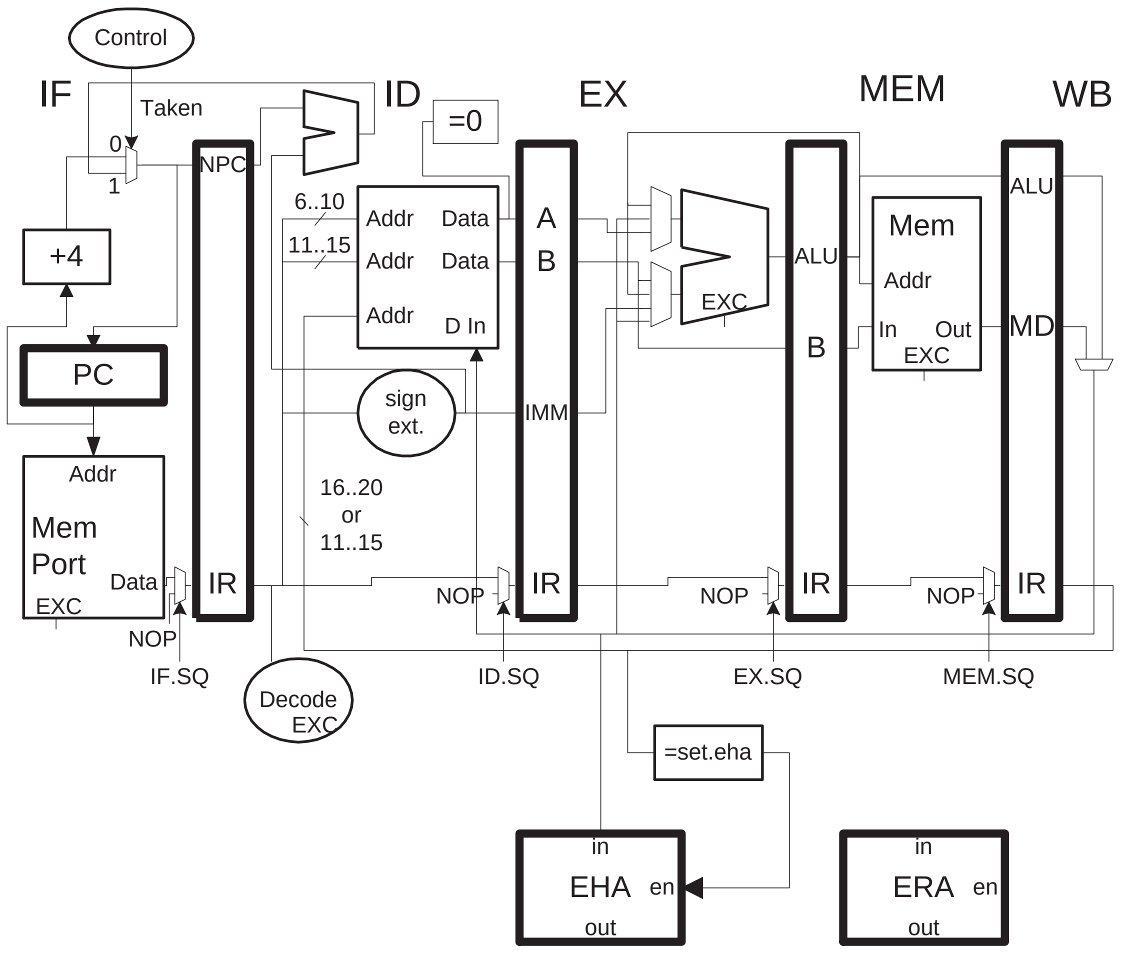
EE 4720 Homework 3 Due: 15 October 1999

Problem 1: Consider the following method of implementing precise exceptions in DLX. An *Exception Handler Address (EHA)* register holds the address of the exception handler and an *Exception Return Address* (ERA) register holds the address of the faulting instruction. A new instruction (not in book) `set.eha` ⟨**rs1**⟩ places the contents of register ⟨**rs1**⟩ in EHA. After an exception occurs the address of the faulting instruction should be put in ERA and control should jump to the address stored in EHA. When an `rfe` (return from exception) instruction is executed control should jump back to the address stored in ERA.

Each stage has a squash signal that effectively replaces any instruction present with a `nop`. (See the illustration below.) Each stage also has an EXC signal which, in the middle of the cycle, is true if an exception is discovered in that stage. EXC will not be asserted if the stage contains an already squashed instruction. Registers EHA and ERA will be written with data at their `in` inputs if `en` is asserted using the same master /slave timing as the other registers and latches.

The diagram below shows a DLX implementation with the new squash signals (IF.SQ, etc.), exception signals (in every stage except WB), and the two new registers. The hardware shown can implement `set.eha` but does not implement exceptions or `rfe`. Add the hardware needed to do these. In particular:

- After an exception occurs control should jump to the address in EHA.
- Exceptions must be precise and handled in program order.
- `rfe` must return control to the faulting instruction.
- If the multiplexor in IF needs additional inputs, use the Taken signal to create the new multiplexor control signal. Taken is asserted only when the ID-stage adder produces the target address.
- Do not implement instructions that transfer ERA to and from an integer register.
- Assume that exception handlers will never encounter exceptions. (They do in real life, so the handler would need a way to save registers before any exceptions occur.)
- Do not test or set processor status bits for privileged state.



Continued on next page.

Based on your design, show a pipeline execution diagram for the code below in which the `lw` instruction raises a page fault exception in MEM and `ant` raises an illegal instruction exception in ID. Show the execution through the first two lines of the handler. Also show execution of the return from the handler and the second call of the handler for the `ant` instruction.

```
lhi  r20, hi(HANDLER)      ! Put high 16 bits in r20.
or   r20, r20, lo(HANDLER) ! Put low 16 bits in r20
set.eha r20                ! In real systems only OS can use this instruction.
add  r1, r2, r3
lw   r4, 0(r5)
ant  r6, r7, r8
sub  r9, r10, r11
and  r12, r13, r15
or   r15, r16, r17

HANDLER:
sw  1000(r0), r1
sw  1004(r0), r2
...
! Return address still in ERA.
lw      r1, 1000(r0)
rfe     ! Handler returns here, code below shouldn't execute.
LINEX:
add    r1, r2, r3
sub    r4, r5, r6
xor    r7, r8, r9
```

In all the problems below all register values are available when the code starts executing. The datapath is fully pipelined so execution of floating point operations can start in the cycle after results are produced, just as the integer instructions do. Unless they are provided, use the following latency and initiation intervals: add unit: latency 3, initiation interval 1; multiply unit: latency 5, initiation interval 1; divide unit: latency 19, initiation interval 20.

Problem 2: Show a pipeline execution diagram for the code below. The branch is **not** taken.

```
 multd f0, f2, f4
 beqz  r1, SKIP   ! Not taken.
 multd f0, f2, f6
 multd f0, f0, f8
 add   r1, r1, r2
```

Problem 3: Show a pipeline execution diagram for the code below. The add functional unit has a latency of 3 and an initiation interval of **2**. *Hint: This problem tests knowledge of initiation intervals, use of functional units by different instructions,* and *usage of registers by single- and double-precision instructions.*

```
LOOP:
 gtd   f12, f14
 addd  f0, f2, f4
 addd  f6, f8, f10
 addf  f16, f7, f18
```

Problem 4: Show a pipeline execution diagram for the code below starting from the first iteration until the CPI for a large number of iterations can be determined. What is that CPI?

The branch condition is bypassed to the ID stage so the branch does not have to stall for `r1`. (See 1998 HW 3.)

```
LOOP:
 subi  r1, r1, #1
 multd f0, f0, f2
 bneq  r1, LOOP
 and   r2, r3, r4
```