Introduction to Digital Computers - Spring 2001
Dr. Guy Even

1 DLX with interrupt handling

This description of interrupt handling in a DLX processor is based on Chapter 8 in the book of Müller and Paul. It is highly recommended that you read that chapter.

Within the text there are "small" questions. These questions are meant to be answered after reading this description to help you see if you understand the text.

1.1 List of interrupts and their properties

Tables 8.1 & 8.2 list the interrupts that we consider. The interrupts are numbered from 0 to 31. Interrupt 0 has the highest priority and interrupt 31 has the lowest priority.

1.2 Signaling an occurrence of an interrupt

The DLX with hardware support has an event signal for every possible interrupt. The event signal is active when the interrupt has occurred. Namely, the occurrence of interrupt i is signaled by a signal $evn[i]$. There is a difference between the signaling of internal interrupts and external interrupts. The following assumptions are used for signaling of interrupts:

External Interrupts: We assume that $env[i]$ is output by a register that belongs to an external device outside of the processor. The external device activates $evn[i]$ when needed. The $evn[i]$ signals remains active until the the Interrupt Service Routine signals to the external device that the interrupt has been serviced and that $evn[i]$ should be deactivated[1].

The reasons for this signaling mechanism are: (a) The control of the DLX may be in any state when $evn[i]$ is activated. We would like the control to be in a unique state (i.e. "fetch") when the decision is made whether an interrupt should be serviced. (b) The task of remembering that an external interrupt is pending is delegated to the external devices (this also simplifies timing issues).

Internal Interrupts: The occurrence of an unmaskable internal interrupt is signaled by a condition of the form:

$$evn[i] = \mathit{States}(i) \ \wedge \ \mathit{Condition}(i)$$

where: (a) $\mathit{States}(i)$ denotes the set of control states in which interrupt i can occur; and (b) $\mathit{Condition}(i)$ denotes the required condition for an occurrence of interrupt i.

Examples:

1. An illegal instruction can occur if: (a) the control is in the "decode" state; and (b) the control reports that the instruction is illegal.

[1]The ISR communicates with the external device simply by loading and storing words in a dedicated area in the memory. For more details, see the topic of *memory mapped I/O* in hardware books

2. A misaligned memory access can occur if: (a) the control is in the states: “fetch”, “load” or “store”; and (b) the memory environment reports a misaligned memory access.

The occurrence of a maskable internal interrupt is signaled by a condition of the form:

$$evn[i] = \textit{States}(i) \ \wedge \ \textit{Condition}(i) \ \wedge \ \textit{SR}(i)$$

where $\textit{SR}(i)$ denotes whether interrupt i is enabled.

Example: An overflow interrupt is signaled if: (a) the control is in an “alu” or “alui” state; (b) the ALU environment is signaling an overflow and the executed instruction is a signed add or subtract instruction; and (c) the overflow interrupt is enabled.

The occurrence of an internal interrupt is signaled only during one clock cycle. The DLX must store this signal so that it can be treated. We will see that the control is modified so that an occurrence of an internal interrupt causes a transition to a special state.

Effect of masking. Interrupt masking has a different effect on internal interrupts and on external interrupts: An occurrence of a disabled internal interrupt is ignored altogether (namely, the occurrence is “lost” forever); an occurrence of a disabled external interrupt is ignored until the interrupt is enabled (i.e. the $evn[i]$ signal remains active).

1.3 Hardware and Software Support

Interrupt handling requires both hardware and software support. The software support is done by a program of the operating system called the *Interrupt Service Routine* (ISR - in short).

1.3.1 Hardware support

Hardware support for handling interrupts consists of four components: special registers, new instructions, extension of the datapath, and extension of the control.

In practice, some of the new instructions are restricted and may only be used by routines that are part of the operating system. This is enforced by defining two modes for the processor: a user mode and a privileged mode. A user program is allowed to use a subset of the instructions that can be used during privileged mode. This means that additional hardware mechanisms need to be introduced to enable changing the mode and checking that restricted instructions are used only in privileged mode. The DLX architecture does not support multiple modes; we simply assume that user programs do not contain restricted instructions.

Special registers. In general, the access to the special registers is restricted and only the operating system is allowed to execute instructions that modify the special registers.

The interrupt handling mechanism uses the following special registers:

1. Status Register $SR[31:0]$. Interrupt i is enabled when $SR[i]=1$. A user can change the contents of the SR register in two ways: (1) Use a system call to enable/disable maskable interrupts. The operating system (i.e. the ISR) can use a privileged instruction "movei2s" to set the contents of the SR. Of course, the system verifies that the request is "legal" before changing the mask. (2) During the execution of an instruction in which an overflow should be signaled (i.e. signed add, signed subtract), the overflow interrupt is enabled.

2. Cause Register $CA[31:0]$. Interrupt i is pending if CA(i) $=1$. The functionality of the cause register CA is given by

$$CA[i](t+1) = \begin{cases} reset \vee pup \vee (CA[0](t) \wedge not(\text{clear\_CA}[t]) & \text{if } i=0 \\ evn[i](t) & \text{if } i \in [31:7] \\ evn[i](t) \vee (CA[i](t) \wedge not(\text{clear\_CA}[t])) & \text{if } i \in [6:1]. \end{cases}$$

   Question: Why and when is the CA register cleared? Who is allowed to clear the CA register?

3. The Exception Program Counter EPC. Holds the return address for the ISR before jumping to the ISR.

4. The Exception Status Register ESR. Holds the contents of the SR before jumping to the ISR.

5. The Exception Cause Register ECA. Holds the list of pending unmasked interrupts when jumping to the ISR. When jumping to the ISR, the ECA register satisfies:

$$ECA[i] = \begin{cases} CA[i] & \text{if } i \leq 6 \\ CA[i[\wedge SR[i] & \text{otherwise} \end{cases}$$

6. The Exception Memory Address Register EMAR. When jumping to the ISR, the contents of the MAR are copied to the EMAR. This enables the ISR that deals with page faults to know which address caused the page fault.

New instructions. Four instructions are used for interrupt handling:

1. `movei2s RS1 SA` : copies the contents of $R_{\langle RS1\rangle}$ to $Special\_R_{\langle SA\rangle}$.

2. `moves2i RD SA` : copies the contents of $Special\_R_{\langle SA\rangle}$ to $R_{\langle RD\rangle}$.

3. `trap type` : a request for a system call. Note that the ISR extracts the type of the system call from the EPC. The trap instruction causes some side effects that are common to the all jumps to the ISR; we will specify these side effects later.

4. `rfe` : return from exception. Restores the SR and PC registers as follows:

$$\begin{aligned} SR &= ESR \\ PC &= EPC \end{aligned}$$

Extension of the datapath. Handling interrupts requires a few modifications in the datapath.

1. PC environment. Previously, the PC register sampled either the new address from the D-Bus or was reset to zero during power-up. The situation now is very similar: during power-up or jump-to-ISR the PC is reset to starting address of the ISR; otherwise, the PC samples the D-bus. Question: when should the PCce signal be active?

2. Memory environment. The write signals are activated only during write accesses which are properly aligned. During a page fault the memory is not changed. The memory environment outputs the event signals for these three interrupts (misaligned access, page fault during fetch, page fault during load or store). The busy signal indicates, as before, whether a successful memory access is not completed yet.

3. Special purpose register environment. Consists of 5 registers (the CA register is considered part of the control). Supports the movei2s and moves2i instructions as well as parallel copying of registers described later.

Extension of the control The following states are added to the control:

1. movs2i, movi2s: these state deal with the execution phase of the corresponding instructions.

2. JISR(-1), JISR1, JISR2: these states deal with part of snap-shooting the state of the processor. We elaborate on these states in Sec. 1.4.1.

3. RFE1, RFE2: these states deal with the execution phase of the rfe instruction. We elaborate on these states in Sec. 1.4.3

External interrupts may occur in any state of the control. The decision to service them takes place only during the "fetch" state.

Internal interrupts occur in specific states of the control (see Table 8.2). A transition to JISR(-1) or JISR1 is added from the states during which an internal interrupt may occur. The occurrence of the corresponding internal interrupt causes a transition to JISR(-1) or JISR1. For example: an occurrence of a page fault during "load" causes a transition to JISR(-1), an occurrence of misaligned access during "store" causes a transition to JISR(-1), an occurrence of an overflow during an "alu" state causes a transition from the write-back states to JISR1. We describe below why and when a transition to JISR(-1) is needed.

The illegal instruction interrupt is an exception to this rule: instead of adding a transition to JISR1, a transition was added to "fetch". The reason for this is simplification of control at the cost of a unified structure (think twice before you apply such tricks...).

The following control signals determine whether jump to the ISR takes place (i.e. whether a transition to JISR(-1) or JISR1 takes place):

1. Interrupt handling is invoked from the "fetch" control state by a signal *int* defined by

$$\mathit{int} = \bigvee_{i=0}^{3} CA[i] \quad \vee \quad \bigvee_{i=7}^{31} (CA[i] \wedge SR[i])$$

   Note that interrupts 0-6 are non-maskable. An illegal instruction (interrupt 1) is detected during "decode", causes the control to return to "fetch" (the default transition in the control is to go to "fetch"), and then $CA[1]$ is checked. Interrupts 4-6 are considered during other control states.

2. From the "load" and "store" control states, interrupt handling is invoked by the signal *intm*, defined by:

$$\textit{intm} = CA[2] \vee CA[4] \quad \text{(misaligned access or page fault)}.$$

3. From write-back control states, interrupt handling is invoked when an unmasked overflow occurs:

$$\textit{int\_ovf} = CA[6].$$

   Note that the masking of the overflow interrupt is already considered when $evn[6]$ is computed.

4. From the "trap" control state, the event signal $evn[5]$ is set, which causes $CA[5]$ to be set as well. The next transition is always to JISR1.

1.3.2 Software support

Software support consists of the Interrupt Service Routine (ISR). The ISR is a DLX program. Correct functionality is obtained by requiring certain properties from the ISR. These requirements are:

1. Every uninterrupted execution of the ISR terminates.

2. The critical sections are not interrupted by interrupts 1-5.

We will see that requirement 2 is satisfied if the following requirements are satisfied:

1. The interrupt stack is stored on a permanent page of memory. Namely, this page is never swapped out, and hence, access to the interrupt stack does not create a page fault.

2. The critical sections of the ISR are stored on a permanent page.

3. The ISRs for interrupts 0-5 are stored on a permanent page.

4. During an execution of the ISRs for interrupts 1-5, illegal instructions, misaligned memory accesses, and page faults do not occur.

1.4 Interrupt Handling

Interrupt handling is divided into three stages, as follows.

1.4.1 Save Status

The first two steps of saving the status are implemented in hardware, the rest are implemented in software.

1. Save the PC of the running program. This PC serves as the return address of the ISR. There are two control states the are involved with this step: JISR1 and JISR(-1), and their actions are:

$$\begin{aligned} \text{JISR(-1)} &: PC \leftarrow PC - 4 \\ \text{JISR1} &: EPC \leftarrow PC \end{aligned}$$

   There is a subtle issue regarding the return address. Recall that the PC is incremented during the "decode" control state. This means that if the "decode" control state has been already executed, then the PC points to the "next" instruction, otherwise it points to the "current" instruction. If the "decode" control state has not been already executed, then the PC points to the "current" instruction. This observation together with whether the interrupt is of type abort/repeat/continue determines which return address should be used, and is reflected in the control in the states JISR1 and JISR(-1).

   In the JISR1 control state, the PC is simply copied to the EPC. However, in the JISR(-1) state, the PC is decremented before it is copied to the EPC (i.e. $EPC \leftarrow PC - 4$) to reverse the increment performed during the "decode" control state so that the return address is the "current" instruction. (Question: check which interrupts go through JISR1 and which interrupts go through JISR(-1)).

2. The JISR1 state is followed by the JISR2 state. During this control state three actions take place simultaneously (i.e. during the same clock cycle): enter a critical section, snapshot the status, and jump to the ISR.

   (a) Enter the critical section. This step consists of two actions:

      i. Disable interrupts 6-31 by setting $SR_{in}[31:6] \leftarrow 0^{26}$. The ISR must be written in such a way that, during the critical section, interrupts $1-5$ do not occur.

      ii. $CA_{in} \leftarrow evn[31:7] \cdot 0^6 \cdot (reset \vee pup)$. This clears the CA and insures that pending internal interrupts 1-6 will not invoke the interrupt handling mechanism during the critical section.

   (b) Snapshot the status:

$$\begin{aligned} ESR &\leftarrow SR_{out} \\ ECA &\leftarrow CA_{out} \wedge SR_{out} \\ EMAR &\leftarrow MAR \end{aligned}$$

   Note that the SR & CA that are used in the snapshot are the *old SR* and *old CA*, namely, the SR and CA before they are cleared during entrance to the critical section! Otherwise, the "cleared" SR is stored and used for masking rather than the "current" SR. We emphasize this by distinguishing between SR_{in} and SR_{out}.

(c) Jump to the ISR: $PC \leftarrow SISR$, where $SISR$ is the fixed address of the beginning of the ISR. (Question: Where does $SISR$ appear in the DLX?)

3. The ISR starts running, and performs the following steps (note, this is software - not hardware!):

   (a) Allocate a new frame for the interrupt on the interrupt stack (in the main memory). This is done by adding $35 \cdot 4$ (i.e 35 words) to the interrupt stack pointer (recall: there are 4 special registers with backup information and 31 general purposes registers). We need an interrupt stack to support nesting of interrupts. A reset interrupt cannot be interrupted (it has the highest priority), and therefore, we don't have to create a new frame if a reset interrupt is being serviced.

   (b) Copy the special registers EPC, ESR, ECA, EMAR to the new frame on the interrupt stack. This is done by using one of the general purpose registers as a "channel" (why?). The instruction movs2i is used to copy a special register to a general purpose register. (How is the value of the general purpose register that is used as a "channel" preserved?). We do this step even if a reset interrupt occurred (why?).

   (c) Compute the pending interrupt with the highest priority:

   $$\ell = \min\{j : ECA[j] = 1\}$$

   (where do we store this value?)

   (d) Exit the critical section. Disable all maskable interrupts with priority lower than ℓ:

   $$SR[i[= \begin{cases} 1 & \text{if } i < \max\{6, \ell\} \\ 0 & \text{otherwise} \end{cases}$$

We are now ready to service interrupt ℓ. (Question: do you know how to program these steps using the DLX instruction set?)

1.4.2 Service the Interrupt

This stage is part of the ISR. The ISR executes the following steps and may be interrupted by an unmasked interrupt.

1. Jump to the segment of the ISR that handles interrupt ℓ.

2. Save the general purpose registers that will be used by the ISR in the interrupt stack.

3. Service the interrupt.

4. Restore the general purpose registers from the interrupt stack.

1.4.3 Restore Status

This stage is also part of the ISR.

1. Enter a critical section by setting: $SR[31 : 6] \leftarrow 0^{26}$.

2. Restore the EPC and the ESR from the interrupt stack. (We will be using a general purpose register for this, and need to restore it from the interrupt stack as well).

3. Free the top frame from the interrupt stack (i.e. subtract $35 \cdot 4$ from the interrupt stack pointer).

4. Exit the critical section. Use the instruction RFE (return from exception) which does the following:

$$\begin{aligned} RFE1: \; SR &\leftarrow ESR \\ RFE2: \; PC &\leftarrow EPC \end{aligned}$$

   The next instruction which is fetched is the instruction which should be executed when returning from the handling the interrupt (this can be an instruction of a user's program or an instruction of an ISR of an interrupt of a lower priority).

An important point to remember is that the critical sections are not interrupted, except for reset interrupts which cannot be "avoided" since they are external. The occurrence of an interrupt of priority 1-5 during a critical section is an error in the ISR.

Questions

1. Why isn't "saving of the general purpose registers" part of "save status"? Similarly, why isn't "restoring the general purpose registers" part of "restore status"?

2. Why isn't the cause register restored during Restore Status?

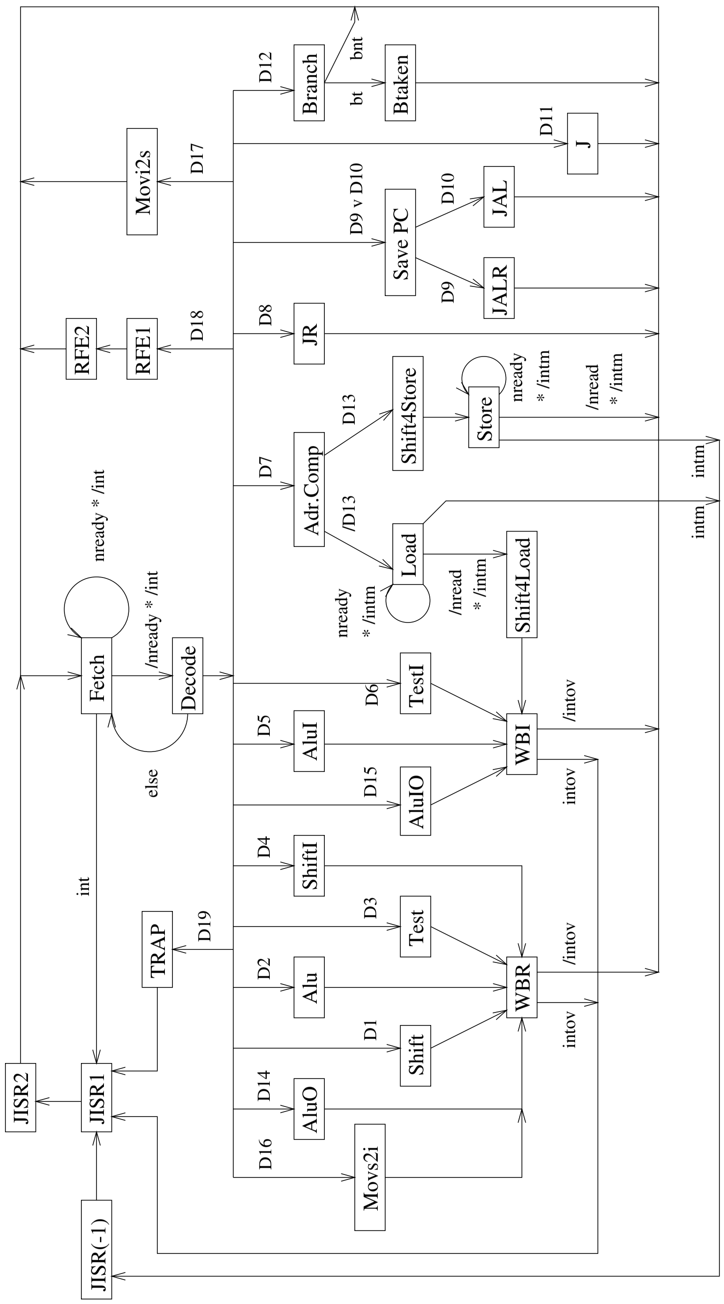

Figure 4: FSD of the DLX design supporting interrupts ($k = 32$ states)