;;;

;;; Initialization code for programs running in the DLX simulator.

;;; Copyright (c) 1999 by Ethan Miller

;;;

;;;

.text

.align 2

;;;----------------------------------------------------------------------

;;; osinit

;;;

;;; This is the first function called by the simulator, even before main.

;;; Since it's called BEFORE any "real" routines, we can use any registers

;;; we want as long as we don't mess up the stack or frame pointers.

;;;

;;; This routine loads the interrupt vector to point to the interrupt

;;; handler later in this file. It should never return, because \_main

;;; should call exitsim() or exit() rather than returning here.

;;;

;;; After initialization is done, this routine jumps to \_main to start the

;;; C code portion of the operating system.

;;;----------------------------------------------------------------------

.proc \_osinit

.global \_osinit

\_osinit:

;; Set up the interrupt handler

lhi r1,(\_intrhandler>>16)&0xffff

addui r1,r1,\_intrhandler&0xffff

movi2s intrvec,r1

;; Never returns because exitsim is called first

j \_main

.endproc \_osinit

;;;----------------------------------------------------------------------

;;; intrhandler

;;;

;;; Called when an interrupt or trap is received by the CPU. It stores the

;;; current register set on the system stack. It then pushes the parameters

;;; to the C interrupt handler onto the stack: the ISR, IAR, and CAUSE

;;; registers along with the original stack pointer. The C interrupt

;;; handler can then copy arguments from the original stack, performing

;;; user -> system translations if necessary.

;;;

;;;----------------------------------------------------------------------

.proc \_intrhandler

.global \_intrhandler

\_intrhandler:

;; We can use r31 as scratch space because its value was saved in

;; ir31. However, we must save the "real" value of r31 on the

;; stack.

;; Always store the registers on the system stack. This way, we don't

;; have to worry about translating things from user -> system. We

;; use r31 as the base register because its value was saved in ir31.

;; If this was a user process, load r29 with the current system

;; stack pointer. If it was a system process, just use the

;; current stack pointer.

movs2i r31,isr

andi r31,r31,0x40

bnez r31,intrSystem

lhi r31,(\_currentPCB>>16)&0xffff

addui r31,r31,\_currentPCB&0xffff

lw r31,(r31)

lw r31,4(r31)

;; Save the original (user) stack pointer

sw -184(r31),r29 ; we haven't yet bumped SP, and 156-340 = -184

;; Copy the system stack pointer into r29 (current stack pointer)

ori r29,r31,0

beqz r0,intrSaveReg ; skip over the system part....

intrSystem:

;; Use the stack pointer we're already using

;; Save r29 because we won't save it later

sw -184(r29),r29 ; we haven't yet bumped SP, and 156-340 = -184

intrSaveReg:

;; Adjust stack pointer for all the stuff we're going to push. This

;; is a bit more space than we need currently, but it leaves room

;; for more stuff if needed.

subui r29,r29,#340

;; Push all the stuff onto the stack

sw 44(r29),r1

sw 48(r29),r2

sw 52(r29),r3

sw 56(r29),r4

sw 60(r29),r5

sw 64(r29),r6

sw 68(r29),r7

sw 72(r29),r8

sw 76(r29),r9

sw 80(r29),r10

sw 84(r29),r11

sw 88(r29),r12

sw 92(r29),r13

sw 96(r29),r14

sw 100(r29),r15

sw 104(r29),r16

sw 108(r29),r17

sw 112(r29),r18

sw 116(r29),r19

sw 120(r29),r20

sw 124(r29),r21

sw 128(r29),r22

sw 132(r29),r23

sw 136(r29),r24

sw 140(r29),r25

sw 144(r29),r26

sw 148(r29),r27

sw 152(r29),r28

;; Skip r29 - stored earlier!

sw 160(r29),r30

;; Load the value of r31 from the special register and then save it

movs2i r3,ir31

sw 164(r29),r3

;; Store the floating-point registers

sd 168(r29),f0

sd 176(r29),f2

sd 184(r29),f4

sd 192(r29),f6

sd 200(r29),f8

sd 208(r29),f10

sd 216(r29),f12

sd 224(r29),f14

sd 232(r29),f16

sd 240(r29),f18

sd 248(r29),f20

sd 256(r29),f22

sd 264(r29),f24

sd 272(r29),f26

sd 280(r29),f28

sd 288(r29),f30

;; NOTE: we don't save the interrupt vector register because it

;; doesn't change from process to process.

;; NOTE: we don't save the status register because most of the flags

;; are the same from process to process if they're in the interrupt

;; handler. Of course, we DO save the ISR.

movs2i r4,iar

sw 296(r29),r4

movs2i r5,isr

sw 300(r29),r5

movs2i r6,cause

sw 304(r29),r6

movs2i r3,fault

sw 308(r29),r3

movs2i r3,ptbase

sw 312(r29),r3

movs2i r3,ptsize

sw 316(r29),r3

movs2i r3,ptbits

sw 320(r29),r3

;; Push the interrupt information onto the stack

sw 0(r29),r6 ; push CAUSE

sw 4(r29),r4 ; push IAR

sw 8(r29),r5 ; push ISR

;; Get the original stack pointer

lw r1,156(r29)

sw 12(r29),r1

;; Save the previous interrupt stack frame address in the current frame

lhi r1,(\_currentPCB>>16)&0xffff

addui r1,r1,\_currentPCB&0xffff

lw r1,(r1)

lw r2,0(r1)

sw 40(r29),r2

;; Save this frame address in the PCB. This is used so the OS can

;; easily access the current interrupt save frame

sw 0(r1), r29

;; Call the "real" interrupt handler. This will possibly switch

;; contexts. This call never returns; instead, a separate routine

;; (\_intrreturn) is called to return from interrupts after restoring

;; the current context.

j \_dointerrupt

nop

.endproc \_intrhandler

;;;----------------------------------------------------------------------

;;; intrreturn

;;;

;;; Return from an interrupt or trap. This restores all of the previously

;;; saved registers and then returns to where the program left off. The

;;; current contents of the registers are destroyed. This routine uses

;;; the saved interrupt frame pointer, so the stack pointer need not

;;; be correct. Note, though, that the register contents from the previous

;;; process must have previously been saved - in other words, call this

;;; routine from a trap or interrupt handler.

;;;----------------------------------------------------------------------

.proc \_intrreturn

.global \_intrreturn

\_intrreturn:

;; Disable interrupts - this routine must be atomic, and interrupts

;; may not be currently disabled. Don't worry about saving registers

;; because we're about to reload them anyway.

jal \_DisableIntrs

;; Get our interrupt stack frame location and load it into the stack

;; pointer.

lhi r1,(\_currentPCB>>16)&0xffff

addui r1,r1,\_currentPCB&0xffff

lw r1,0(r1)

lw r29,0(r1)

;; Get the previous interrupt stack frame location and make it the

;; current interrupt save frame.

lw r2,40(r29)

sw 0(r1), r2

;; Reload the registers for the new process. We don't have to

;; load in the exact opposite order as long as we're careful to

;; get the right values back in.

lw r3,296(r29)

movi2s iar,r3

lw r3,300(r29)

movi2s isr,r3

lw r3,304(r29)

movi2s cause,r3

lw r3,308(r29)

movi2s fault,r3

lw r3,312(r29)

movi2s ptbase,r3

lw r3,316(r29)

movi2s ptsize,r3

lw r3,320(r29)

movi2s ptbits,r3

;; Reload the floating point registers

ld f0,168(r29)

ld f2,176(r29)

ld f4,184(r29)

ld f6,192(r29)

ld f8,200(r29)

ld f10,208(r29)

ld f12,216(r29)

ld f14,224(r29)

ld f16,232(r29)

ld f18,240(r29)

ld f20,248(r29)

ld f22,256(r29)

ld f24,264(r29)

ld f26,272(r29)

ld f28,280(r29)

ld f30,288(r29)

;; Reload the integer registers. We don't reload r0 because it's

;; always 0. We won't reload r29 here because we're using it as

;; the stack pointer. The same goes for r1, which we'll use as

;; scratch so we can store r29.

;; Skip r1 - restored later

lw r2,48(r29)

lw r3,52(r29)

lw r4,56(r29)

lw r5,60(r29)

lw r6,64(r29)

lw r7,68(r29)

lw r8,72(r29)

lw r9,76(r29)

lw r10,80(r29)

lw r11,84(r29)

lw r12,88(r29)

lw r13,92(r29)

lw r14,96(r29)

lw r15,100(r29)

lw r16,104(r29)

lw r17,108(r29)

lw r18,112(r29)

lw r19,116(r29)

lw r20,120(r29)

lw r21,124(r29)

lw r22,128(r29)

lw r23,132(r29)

lw r24,136(r29)

lw r25,140(r29)

lw r26,144(r29)

lw r27,148(r29)

lw r28,152(r29)

;; Skip r29 - restored later

lw r30,160(r29)

lw r31,164(r29)

addui r29,r29,#340

;; Save the current value of the stack pointer after adjusting it

;; Note that this will "destroy" the stack values below this interrupt

;; stack frame. This is exactly what we want!

sw 4(r1),r29

ori r1,r29,#0

lw r29,-184(r1) ; 156-340 = -184

lw r1,-296(r1) ; 44-340 = -296

rfe

.endproc \_intrreturn

;;;----------------------------------------------------------------------

;;; SetIntrs

;;;

;;; This routine sets the interrupt level to the value passed (0 -> all

;;; interrupts enabled; 0xf -> all interrupts disabled). It returns the

;;; former value for the interrupt flags.

;;;----------------------------------------------------------------------

.proc \_SetIntrs

.global \_SetIntrs

\_SetIntrs:

subui r29,r29,#16

sw 12(r29),r2 ; save r2

lw r2,16(r29) ; Get the new interrupt level

andi r2,r2,#0x0f ; Mask off interrupt levels

movs2i r1,status

sw 8(r29),r1 ; Store the old interrupt values

andi r1,r1,#0xfff0 ; Mask off old interrupt level

or r1,r2,r1 ; OR in new interrupt level

movi2s status,r1

lw r1,8(r29) ; Get back the original interrupt level

andi r1,r1,#0x0f ; Mask off all but interrupt levels

lw r2,12(r29) ; restore r2

addui r29,r29,#16 ; restore stack pointer

jr r31

nop

.endproc \_SetIntrs

.proc \_CurrentIntrs

.global \_CurrentIntrs

\_CurrentIntrs:

movs2i r1,status

andi r1,r1,#0xf

jr r31

nop

.endproc \_CurrentIntrs

;;;----------------------------------------------------------------------

;;; \_ProcessSleep

;;;

;;; If a context switch from elsewhere in the kernel is desired, take a

;;; trap and call this routine from the trap handler.

;;;----------------------------------------------------------------------

.proc \_ProcessSleep

.global \_ProcessSleep

\_ProcessSleep:

trap #0x410 ; This is a process sleep trap

nop

jr r31

nop

.endproc \_ProcessSleep