I/O, Peripherals, HW/SW Interface

Forrest Brewer
Chandra Krintz

I/O Device Interface

- Interface is composed of:
- Driver (SW), bus and HW peripheral, physical HW device
- Overall, the interface introduces an abstraction layer (or several) simplifying the process of making use of the physical (or virtual) device.
  - File I/O in linux/windows looks like byte stream or record I/O
  - Network officially uses 7 layers of abstraction
- Typically:
  - Driver implements:
    - Device Initialization and Reset
    - Initialization of Data Transfers and Management of Data Flow
    - Device Shutdown and Removal
  - Often two driver interfaces: data channel and device control

I/O device: software side

- Memory Map
  - Hardware Glue is used to create a physical address space which is serviced by dedicated hardware as if it was memory
  - Conceptually Simple from program viewpoint
  - Potentially Breaks Memory Paradigm
    - Memory values change w/o CPU activity…
- I/O port
  - Use two or more address spaces
  - Ports can be written or read, but are not Memories
  - Sometimes special I/O instructions or Status Bit
- Issues
  - Kernel time limited – but
    - CPU protection modes
    - Coherence atomicity
  - Preservation of Device State
    - Minimal Kernel state for synchronization/modal behavior

I/O Device: hardware side

- Physically must decode Memory address bus or I/O port address, then manage physical data transfer to device
  - Data formats and rates may be **very** different
    - EG SPI based A/D is a bit-serial interface with rates between 20kHz and 50+MHz, yet CPU is expecting a Byte or Word parallel transfer on its event timing.
    - Physical device usually has own idea about time and who is boss…
    - Usually CPU is forgiving about adding ‘wait’ states or delaying trasnfers
  - Device operation timescales can be much faster or much slower than CPU software events, yet a reliable, efficient interface is needed.
    - Minimal: HW synchronization from event to CPU Bus
    - Often, buffering FIFO and interrupt generation as well as protocol

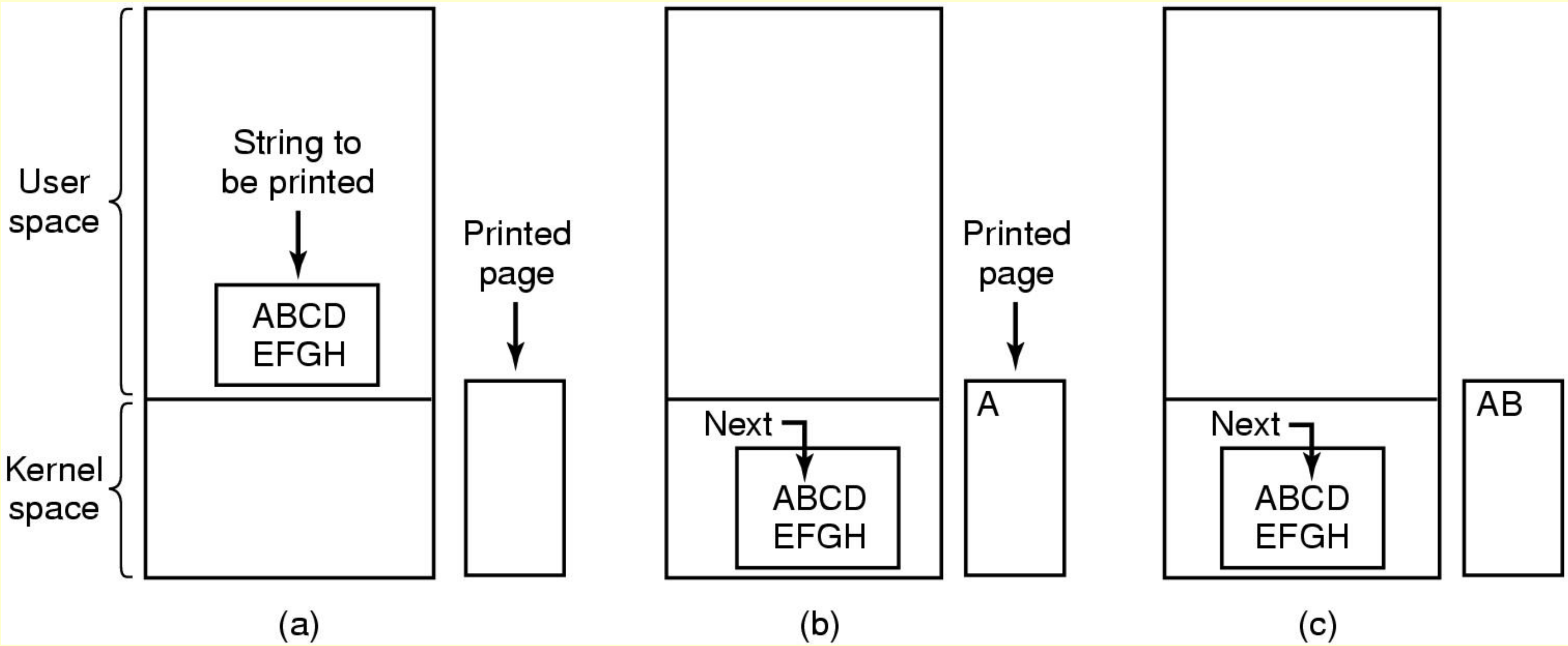
Efficient Interfacing

- Service dozens of peripherals, each with own time scale
- How to keep data transfers coherent?
  - How to prevent slow devices from slowing down system?
- Classically, two kinds of Interface
  - Polling (Program Driven I/O)
    - CPU polls the device addresses and takes action when needed
    - Simple to build HW, but CPU needs to poll often – so may not be efficient if lots of devices
    - Sequential program flow is maintained
  - Interrupts (Event Driven I/0)
    - Set up event, then go off and do other things until signaled
    - On signal, drop everything, service need and resume other things
    - Allows for preempt of CPU as events dictate, but
    - Breaks sequential program flow

Buses

- Shared communication link (one or more wires)
- Types of buses:
  - processor-memory (short, high speed, DDR, LVDS)
  - backplane (longer, high speed, PCI, PCIe, ISA, PLB)
  - I/O (longer, different devices, e.g., USB, Firewire)
  - Network (Very long, standardized e.g. Internet, Phone..)
- Bus length refers to
- MicroBlaze supports PLB, OPB, MLB, SPI, I^2C…
  - Each needs hardware physical peripheral and
  - Software device driver
- Synchronous vs. Asynchronous
  - Practically all buses are somewhat Asynchronous but
  - Simulate synchronous behavior to avoid rendezvous signals

Programmed (Polled) I/O



Steps in printing a string

Programmed I/O Example

- Writing a string to a (RS-232) serial output

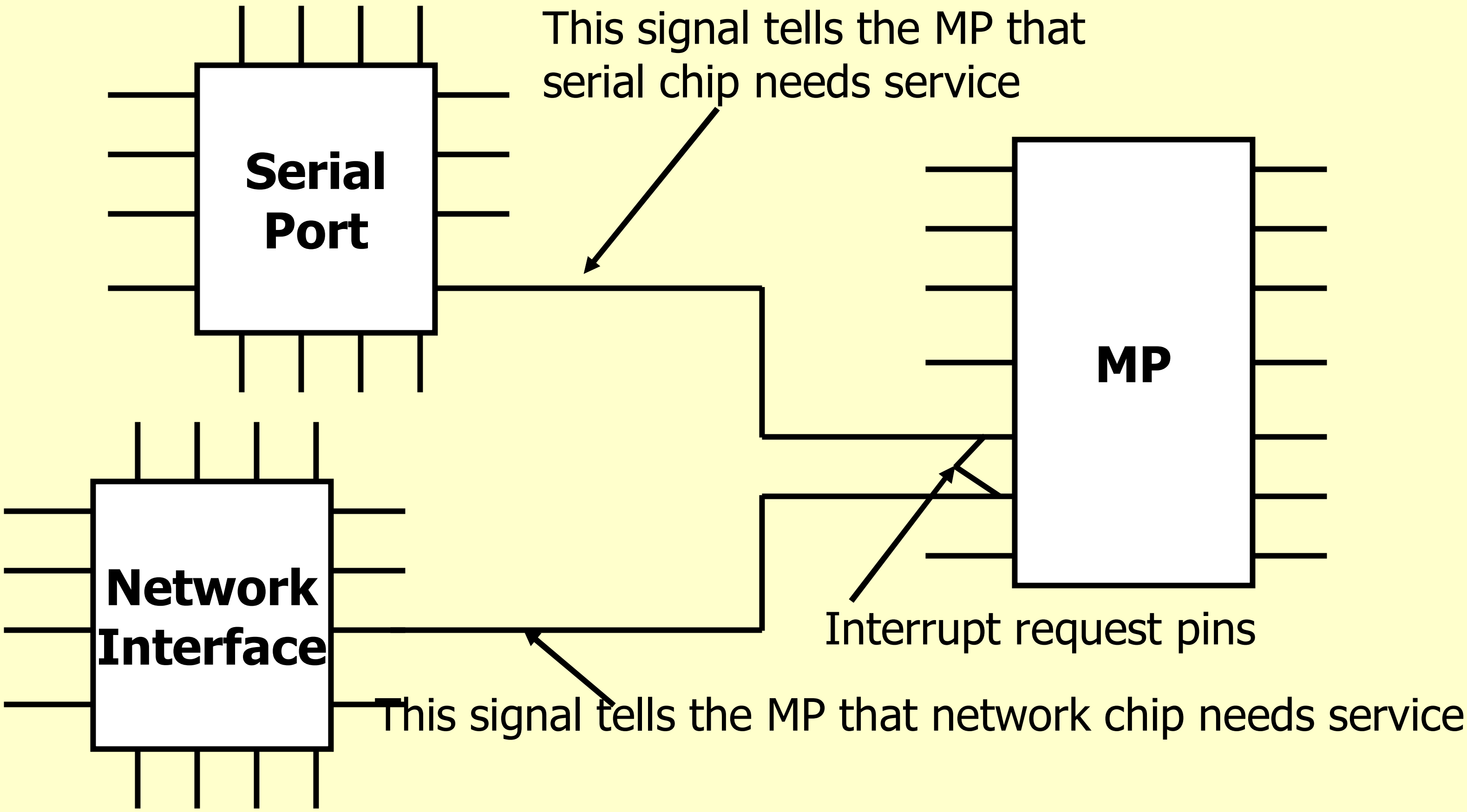
```
CopyFromUser(virtAddr, kernelBuffer, byteCount);
for(i = 0; i<byteCount; i++) {
    while (*serialStatusReg != READY);
    *serialDataReg = kernelBuffer[i];
}
return;
```

- Called “Busy Waiting” or “Polling”
- Simple and reliable, but CPU time and energy waste
- This is what happens when you talk to slow devices (like Board LCD) with 1 control thread…

Better Programmed I/O

- Idea– don't wait locally for events, doing nothing else
- Instead, poll for multiple events by merging local loops into larger one.
  - Leads to 'grand loop' designs
  - Works only if devices are slow compared to CPU
- If devices are really slow– wastes CPU power
- Can be generalized if you know something about the pattern of arriving signals.
- Maybe better idea is to use hardware to do the 'scan' for change of I/O state?

Interrupt Signalling



Interrupt-Driven I/O

- Getting the I/O started:

```
CopyFromUser(virtAddr, kernelBuffer, byteCount);
EnableInterrupts();
i = 0;
while (*serialStatusReg != READY);
*serialDataReg = kernelBuffer[i++];
sleep ();
```

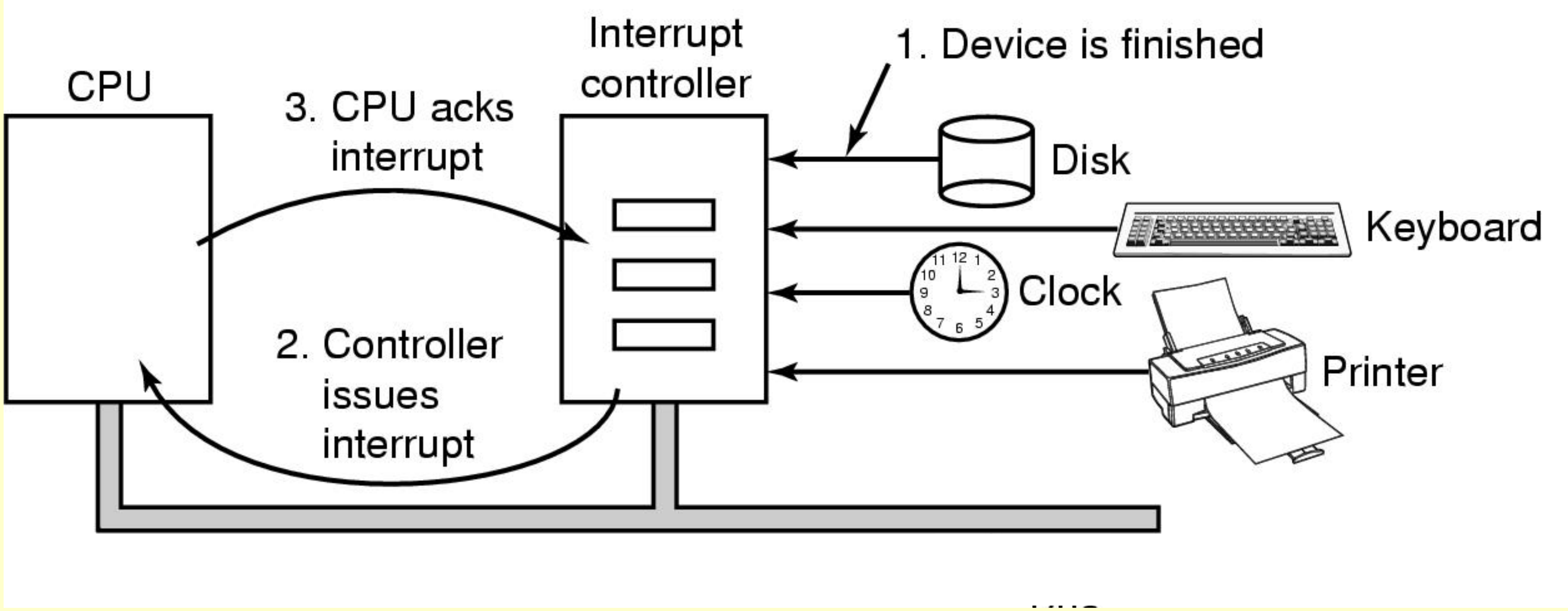
- The Interrupt Handler:

```
if (i == byteCount)
  Wake up the user process
else{
  *serialDataReg = kernelBuffer[i]
  i++;
}
Return from interrupt
```

Lifetime of an Interrupt

- External hardware signals request
  - here, device signals that data in serialStatusReg has been sent
- CPU
  - Checks if interrupt can be taken
  - Jumps to interrupt handler
  - Executes handler
  - Returns to interrupted task

Hardware support for interrupts



If there are many devices, an interrupt controller can do the work instead of the CPU using multiple I/O pins.

Connections between devices and interrupt controller actually use interrupt lines on the bus rather than dedicated wires

Interrupt driven I/O Issues

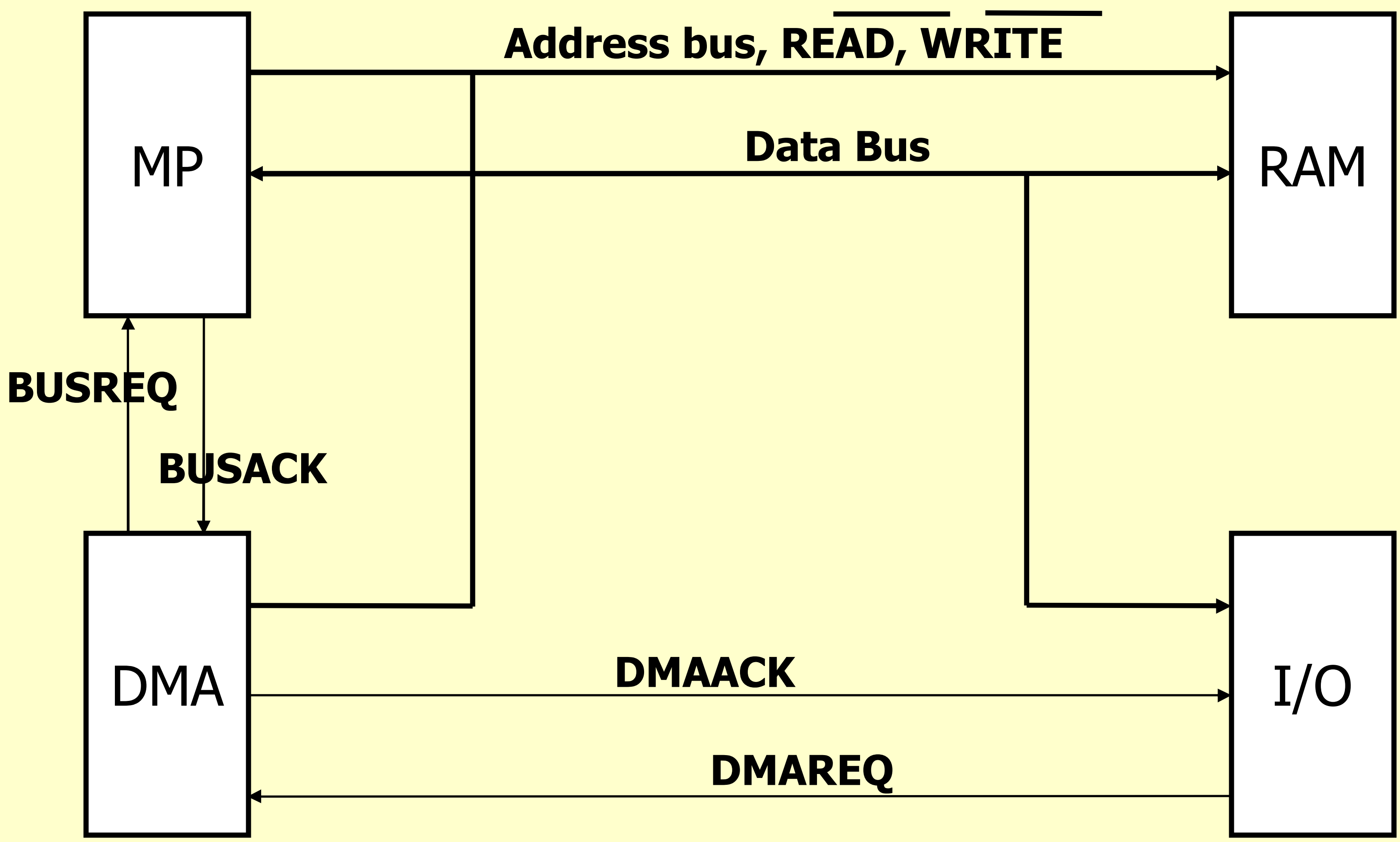
- Problem:
  - CPU is still involved in every data transfer
  - Interrupt handling overhead is high
  - Overhead cost is not amortized over much data
  - Overhead is too high for fast devices
    - Gbps networks
    - Disk drives

Direct Memory Access

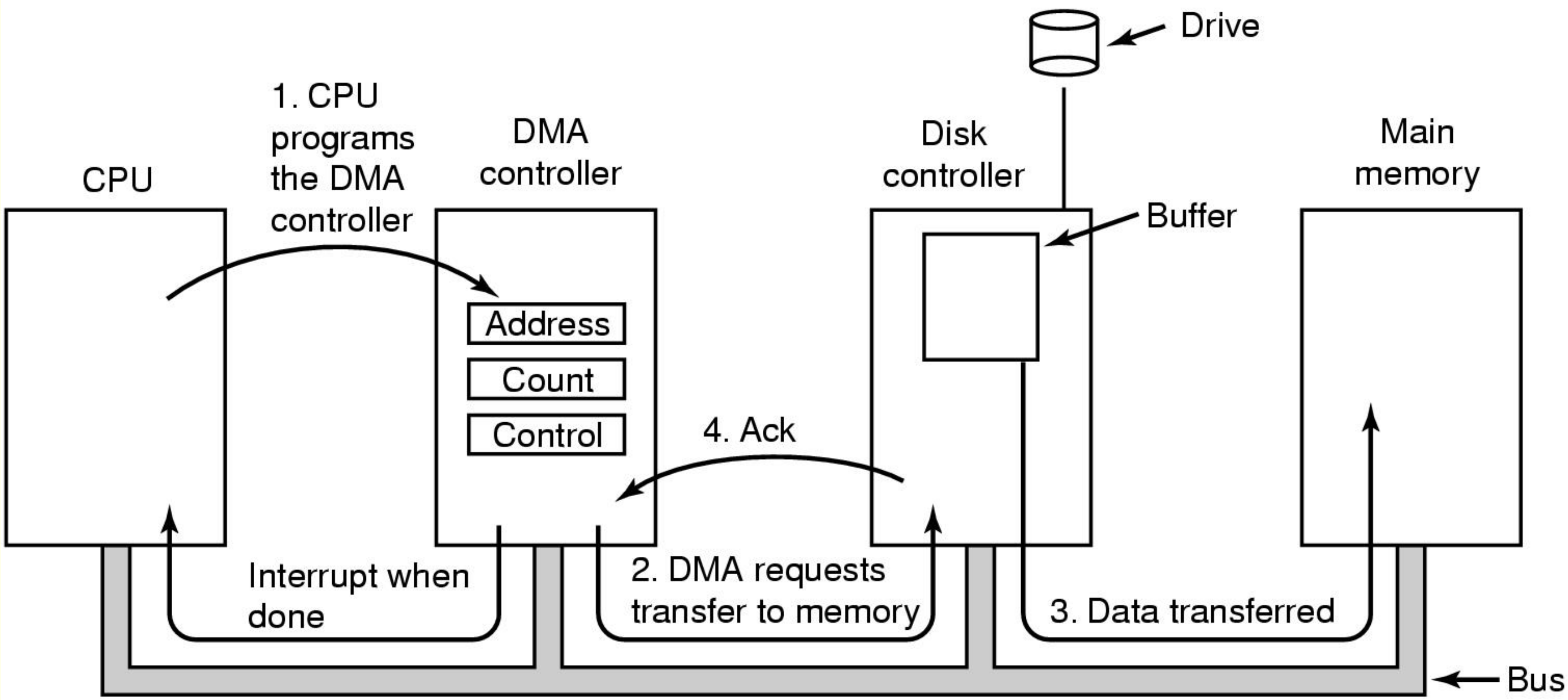
- Get data in and out of systems quickly
- Direct Memory Access (DMA)
  - Reads data from I/O devices, writes it to memory
  - Reads data from memory, writes it to the I/O device
  - Without software and MP intervention
  - i.e. Very simple ancillary processor

- Potential problems
  - Must not interfere with MP on the bus (address/data lines)
    - Often does, of course– idea is to keep the overhead low…

DMA


MP
RAM
DMA
I/O
Address bus, READ, WRITE
Data Bus
BUSREQ
BUSACK
DMAACK
DMAREQ

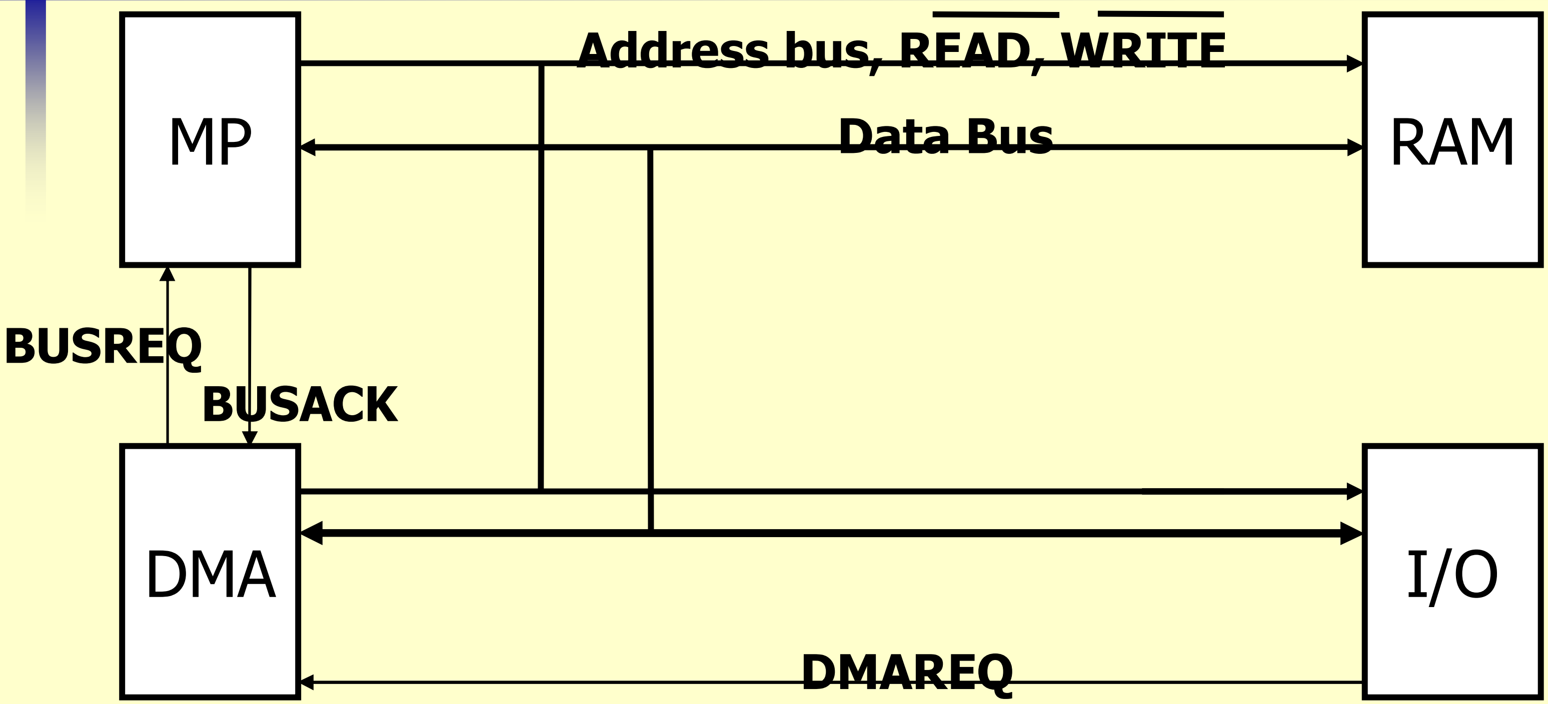
Direct Memory Access (DMA)



DMA

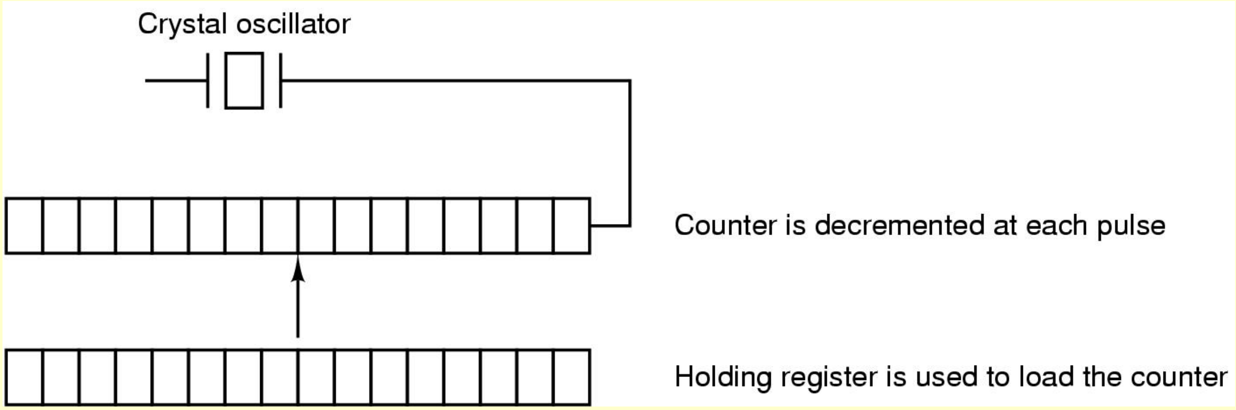
- How does the DMA know to transfer additional bytes after the first has been transferred?
- Edge triggered
  - DMA transfers a byte each time it sees a rising DMAREQ edge
    - I/O must **re-raise** (possibly immediately) DMAREQ for each byte
    - CPU gets control back between bytes
    - Inefficient transfer though
- Level triggered
  - Burst mode
  - DMA transfers bytes as long as DMAREQ is high
    - I/O must lower DMAREQ **explicitly** when it is done
    - CPU without bus for longer periods
    - Interrupt service may not be timely
- DMA can have other implementations

DMA (Alternative Architecture)



- When I/O wants to write to memory, it instead writes to DMA (internal register storage)
  - Simplifies I/O device, makes DMA device more complicated
  - Transfer time doubles since two bus transfers are being performed

Sample I/O Devices: Programmable clocks



- One-shot mode:
  - Counter initialized then decremented until zero
  - At zero a single interrupt occurs
- Square wave mode:
  - At zero the counter is reinitialized with the same value
  - Periodic interrupts (called “clock ticks”) occur

Time

- <86 seconds to exhaust 32-bit at 50MHz
  - How can we remember what the time is?
  - 64-bit clock is good for >11,000 years…
- Backup clock
  - Similar to digital watch
  - Low-power circuitry, battery-powered
  - Periodically reset from the internet
  - UTC: Universal Coordinated Time
  - Unix: Seconds since Jan. 1, 1970
  - Windows: Seconds since Jan. 1, 1980

Goals of embedded clocks

- Prevent processes from dominating CPU
- Timestamp external events (such as A/D conversion events).
  - Can provide hardware register with value of clock at reporting (or interrupt) time of device
  - Can correct sample value by interpolation to expected value at desired (not quite real) sample time.
- Provide for timely task or process switching
- Provide event timing for external devices

Interrupts and Interrupt Handling

Interrupt Request (IRQ)

- When an IRQ is asserted
  - MP stops doing what it was doing (executing instructions)
    - **Completes** execution of the instruction that is executing
    - flush the instructions currently pending execution
  - Create new stack frame (after any required context switch)
    - Saves the next address on the stack
    - Like a return address when a CALL instruction is executed
    - However the `CALL' is done automatically
  - Jumps to **interrupt routine**
    - Interrupt handler or service routine (ISR)
    - Very **short, fast** subprogram
    - Interrupts live in real-time, often on system SW

Interrupt Routine/Handler

- Interrupt handler, Interrupt service routine (ISR)
  - Very short, fast code
  - Implemented like a subprogram
  - **All used** registers must be saved and restored
    - Saving the **context**
    - **Handled by _interrupt_handler_ function attribute**
  - **Any latency in service routine shows up in every event response.**

Interrupt Routines

- Notice: Interrupt can occur between **any** two instructions
  - CALL instruction: compiler knows what code came before and after the call
  - Compiler can write code to save/restore registers used in the callee
- The compiler (when generating code for interrupt handler)
  - Or assembly programmer
  - Can not know when interrupts will occur
  - Therefore **ALL** non-volatile registers used by the interrupt handler must be saved and restored to ensure register preservation

Disabling Interrupts

- Every system allows interrupts to be disabled in some way
  - Devices can be told not to interrupt
  - MP can be told to ignore interrupts
  - In addition, the MP can ignore some subset of interrupts
- Commonly individual interrupts can be disabled
- If an interrupt occurs while turned off, the MP remembers
  - Deferred, not really disabled
  - Runs immediately after return from interrupt

Disabling Interrupts

- Nonmaskable interrupt
  - An interrupt that cannot be turned off **ever**
  - Used for exceptional circumstances
    - Beyond the normal range of ordinary processing
    - Catastrophic event
    - Power failure
  - How do you reset Mars Rover if gets stuck in an infinite loop?

Interrupt Nesting

- When an interrupt occurs while an interrupt handler is executing
  - For some systems, this is the default behavior
    - When priorities are used - **only** a higher priority interrupt can interrupt the handler
    - If a lower priority IRQ is raised, the current handler completes before the low-priority IRQ is handled
  - For others, special instructions are inserted into interrupt routines to indicate that such behavior can occur
    - For this case, all interrupts are automatically disabled whenever an interrupt handler is invoked
    - MicroBlaze chose this way…

Interrupt Priorities

- Each interrupt request signal (IRQ) can be assigned a **priority**
- Programs can set the lowest priority it is willing to accept
  - By setting this priority higher, a program effectively **disables** all interrupts below this priority
  - Most systems use prioritized interrupts and allow individual interrupts to be turned off
- When multiple IRQs are raised at the same time
  - MP invokes the handler of each according to (highest) priority

Worst-Case Interrupt Response

- Consider a system with 3 levels of interrupt priority:
  - 400uS latency in highest priority level handler
  - 4mS latency in middle priority level handler
  - 10mS latency in lowest priority level handler
- What is worst case response time for highest priority interrupt?
  - Might guess 400uS– but…

  If system just started servicing a low priority interrupt, interrupts are disabled so scheduler cannot alter program flow until end of handler. So high priority handler completion could happen after 10.4mS…
- Worst case response time for middle priority interrupt?
  - Forever!

Interrupt Handlers Code Location

- Commonly, a table is used
  - Holds addresses of interrupt handler routines
    - **Interrupt vectors**
  - Called an interrupt vector table
  - Indexed by a unique number assigned to each interrupt
  - Rarely changes - implemented with crt0.c initialization
  - Location
    - Known/fixed location
    - Variable location w/ known mechanism for telling the MP where it is
- When an IRQ is raised
  - MP looks up the interrupt handler in the table
  - Uses the address for to branch to handler
  - Handler lookup and dispatch can be part of interrupt or can be done in hardware

Interrupt-Driven I/O

- Getting the I/O started:

```
CopyFromUser(virtAddr, kernelBuffer, byteCount);
EnableInterrupts();
i = 0;
while (*serialStatusReg != READY);
*serialDataReg = kernelBuffer[i++];
sleep ();
```

- The Interrupt Handler:

```
if (i == byteCount)
  Wake up the user process
else{
  *serialDataReg = kernelBuffer[i]
  i++;
}
Return from interrupt
```

Lifetime of an Interrupt

- External hardware signals request
  - here, device signals that data in serialStatusReg has been sent
- CPU
  - Checks if interrupt can be taken
  - Jumps to interrupt handler
  - Executes handler
  - Returns to interrupted task

Interrupt Nesting

- When an interrupt occurs while an interrupt handler is executing
  - For some systems, this is the default behavior
  - When priorities are used - **only** a higher priority interrupt can interrupt the handler
    - If a lower priority IRQ is raised, the current handler completes before the low-priority IRQ is handled
  - For others, special instructions are inserted into interrupt routines to indicate that such behavior can occur
    - disable(); enable(); disable(id); enable(id);
    - For this case, all interrupts are automatically disabled whenever an interrupt handler is invoked
      - **Unless the instructions are present which re-enables interrupts**

Shared Data Problem

- Very little should be done in the interrupt handler
  - To ensure that interrupts are handled quickly
  - To ensure that control returns to the task code ASAP

- Interrupt routine must tell task code to do followup processing
  - To enable this, the interrupt routine and the task code communicate using shared variables.

Shared Data Problem

- Interrupt routine must tell task code to do followup processing
  - Via shared memory (variables)
    - Used for communication between handler and task code
    - **Shared**
  - However, shared data is a well-known, difficult problem
    - Handlers-tasks
    - Across tasks
    - Across threads
  - Because **two+** entities are **interleaved** - and each can modify the same data!

Nuclear Reactor Monitoring System

- Monitors two temperatures that must always be equal

```
static int iTemperatures[2];
void interrupt vReadTemperatures () {
   iTemperatures[0] = //read value from hardware
   iTemperatures[1] = //read value from hardware
}
void main() {
   int   iTemp0,   iTemp1;
   while(TRUE) {
      iTemp0 = iTemperatures[0];
      iTemp1 = iTemperatures[1];
      if (iTemp0 != iTemp1) {
             //set off alarm!
      }
   }
}
```

Interrupt can occur between any two instructions!

Nuclear Reactor Monitoring System

```
static int iTemperatures[2];
void interrupt vReadTemperatures () {
   iTemperatures[0] = //read value from hardware
   iTemperatures[1] = //read value from hardware
}
void main() {
   int  iTemp0,  iTemp1;
   while(TRUE) {
      iTemp0 = iTemperatures[0];
      iTemp1 = iTemperatures[1];
      if (iTemp0 != iTemp1) {
            //set off alarm!
      }
   }
}
```

Problem!

Interrupt can occur between any two instructions!

Nuclear Reactor Monitoring System

```
static int iTemperatures[2];
void interrupt vReadTemperatures () {
    iTemperatures[0] = //read value from hardware
    iTemperatures[1] = //read value from hardware
}

void main() {
    while(TRUE) {
      if (iTemperatures[0] != iTemperatures[1]) {
              //set off alarm!
      }
    }
}
```

```
static int iTemperatures[2];
void interrupt vReadTemperatures () {
    iTemperatures[0] = //read value from hardware
    iTemperatures[1] = //read value from hardware
}

void main() {
    while(TRUE) {
      if (iTemperatures[0] != iTemperatures[1]) {
            //set off alarm!
      }
    }
}
      PUSH    AR3
      LDIU    SP, AR3
L11   LDIU    @2E2h, R0
      CMPI    @2E3h, R0
      BZ      L11
      // code in if part of the branch (executed when not taken)
      BU      L11
```

Same Problem!

Interrupt can occur between any two instructions!

Shared Data Problem

- Difficult because
  - They don't happen every time the code runs
    - In the previous example, the interrupt could happen at other points in main's execution and doesn't cause a problem
  - Events (interrupts) happen in different orders at different times
    - Non-deterministic (not necessarily repeatable)

- Possible solution 1
  - Disable interrupts whenever the task code uses shared data
- Possible Solution 2
  - Perform Comparison in interrupt routine (when interupts are disabled)
    - Can be cagey about which calls to the service routine do it…

```
static int iTemperatures[2];
void interrupt vReadTemperatures () {
    iTemperatures[0] = //read value from hardware
    iTemperatures[1] = //read value from hardware
}
void main() {
    int   iTemp0,   iTemp1;
    while(TRUE) {
      disable();
      iTemp0 = iTemperatures[0];
      iTemp1 = iTemperatures[1];
      enable();
      if (iTemp0 != iTemp1) {
            //set off alarm!
      }
    }
}
```

Atomic Code

- Shared-data problem
  - Task code and interrupt routine share data
  - **Task code uses the data in a way that is not atomic**
  - Solution: Disable interrupts for task code that uses data
  - **Atomic code:** Code that cannot be interrupted
- **Critical section**: set of instructions that must be atomic for correct execution
- Atomic code
  - Code that cannot be interrupted **by anything that might modify the data being used**
  - Allows specific interrupts to be disabled as needed
    - And others left enabled

```
static int iSeconds, iMinutes, iHours;
void interrupt vUpdateTime() {
    if (++iSeconds >= 60) {
        iSeconds = 0;
         if (++iMinutes >= 60) {
             iMinutes = 0;
             if (++iHours >= 24) {
                  iHours = 0;
             }
         }
     }
    // update the HW as needed
}

long lSecondsSinceMidnight() {
  return(
  (((iHours*60)+iMinutes)*60)+iSeconds);
}
```

Hardware timer asserts an IRQ each second

- Invoking vUpdateTime
- Where is the problem?

```
static int iSeconds, iMinutes, iHours;
void interrupt vUpdateTime() {
    if (++iSeconds >= 60) {
        iSeconds = 0;
         if (++iMinutes >= 60) {
             iMinutes = 0;
      if (++iHours >= 24) {
          iHours = 0;   }
        }
    }
    // update the HW as needed
}

long lSecondsSinceMidnight() {
  disable();
  return(
  (((iHours*60)+iMinutes)*60)+iSeconds;
  enable();
}
```

- Hardware timer asserts an IRQ each second
  - Invoking vUpdateTime
  - **Is this a solution?**

```
static int iSeconds, iMinutes, iHours;
void interrupt vUpdateTime() {
    if (++iSeconds >= 60) {
        iSeconds = 0;
         if (++iMinutes >= 60) {
             iMinutes = 0;
      if (++iHours >= 24) {
          iHours = 0;   }
        }
    }
    // update the HW as needed
}

long lSecondsSinceMidnight() {
 disable();
  long retn =
  (((iHours*60)+iMinutes)*60)+iSeconds;
 enable();
 return retn;
}
```

- Hardware timer asserts an IRQ each second
  - Invoking vUpdateTime
  - ***Is this a solution?***

```
static int iSeconds, iMinutes, iHours;
void interrupt vUpdateTime() {
    if (++iSeconds >= 60) {
        iSeconds = 0;
         if (++iMinutes >= 60) {
             iMinutes = 0;
      if (++iHours >= 24) {
          iHours = 0;   }
        }
    }
    // update the HW as needed
}

long lSecondsSinceMidnight() {
 disable();
  long retn =
 (((iHours*60)+iMinutes)*60)+iSeconds;
 enable();
 return retn;
}
```

- Hardware timer asserts an IRQ each second
  - Invoking vUpdateTime
  - **Is this a solution?**
  - What happens if lSecondsSinceMidnight() is called from within a critical section?:

disable();

. . .

lSecondsSinceMidnight();

. . .

enable();

- **Check before disabling**
  - **Disadvantages?**

Interrupt Latency

- Interrupt response time (**per interrupt**):
  1. Longest period of time (this/all) interrupts are disabled
  2. Execution time for any interrupt routine that has higher **(or equal)** priority than the currently executing one
     1. **Assumes that each one only executes once!**
  3. Context switching overhead by the MP
     1. Time to sense the IRQ, complete the currently executing instruction(s), build stack frame and invoke handler
  4. Execution time of the current interrupt routine to the point that counts as a "response"
- Reducing response time
  - Short handlers, short disable time

Interrupt Latency

- Disabling interrupts
  - Doing it often, increases your response time
- Real-time Systems require guarantees about response time
  - As you design the system you must ensure that such guarantees (real-time or not) are met
- Often, you can avoid disabling interrupts
  - Via careful coding, but…
  - Makes code fragile
  - Difficult to ensure that you’ve got it right

Interrupt Latency Example

- Task code disable time
  - 125 μs to read temp values (shared with temp. hw)
  - 250 μs to read time value (shared with timer interrupt)

- Interprocessor interrupt
  - Another processor causes an IRQ
    - System must respond in 650 μs
    - Handler requires 300 μs

- Worst case wait response time for interprocessor IRQ
  - Handler routine (300 μs)
  - Longest period interrupts are disabled (250 μs)
  - 550 μs - meets (barely) the response requirement

Interrupt Latency Example II

- Task code disable time
  - 125 μs to read temp values (shared with temp. hw)
  - 250 μs to read time value (shared with timer interrupt)
- Interprocessor interrupt
  - Another processor causes an IRQ
    - System must respond in 650 μs
    - Handler requires 300 μs
- Network device
  - Interrupt routine takes 150 μs
- Worst case wait response time for interprocessor IRQ?
  - Will interprocessor interrupt deadline be met?
- Can you improve on this?

Interrupt Latency Example

- Task code disable time
  - 125 μs to read temp values (shared with temp. hw)
  - 250 μs to read time value (shared with timer interrupt)
- Interprocessor interrupt
  - Another processor causes an IRQ
    - System must respond in 650 μs
    - Handler requires 300 μs
- Network device
  - Interrupt routine takes 150 μs
- Worst case wait response time for interprocessor IRQ? 700uS
  - Will interprocessor interrupt deadline be met? no
- Can you improve on this? Make network dev. Lower prty.

Interrupt Latency Example

- Task code disable time
  - 125 μs to read temp values (shared with temp. hw)
  - 250 μs to read time value (shared with timer interrupt)
- Interprocessor interrupt
  - Another processor causes an IRQ
    - System must respond in 650 μs
    - Handler requires **350** μs
- Network device
  - Interrupt routine takes 100 μs
  - Lower priority than interprocessor interrupt
- Worst case wait response time
  - For the interprocessor IRQ?
  - For the network device?

Interrupt Latency Example

- Task code disable time
  - 125 μs to read temp values (shared with temp. hw)
  - 250 μs to read time value (shared with timer interrupt)
- Interprocessor interrupt
  - Another processor causes an IRQ
    - System must respond in 650 μs
    - Handler requires **350** μs
- Network device
  - Interrupt routine takes 100 μs
  - Lower priority than interprocessor interrupt
- Worst case wait response time
  - For the interprocessor IRQ? 600
  - For the network device? 700

Interrupt Latency Example

- Task code disable time
  - 125 μs to read temp values (shared with temp. hw)
  - 250 μs to read time value (shared with timer interrupt)
- What happens if two disable periods happen back to back?
  - How to avoid?

Volatile Variables

- Most compilers assume that a value in memory never changes unless the program changes it
  - Uses this assumption to perform **optimization**
  - However, interrupts can modify shared data
    - Invalidating this assumption
  - Holding memory values in registers is a problem
- An alternate solution to disabling interrupts is to identify single piece of data that is shared with interrupt routines
  - Make sure that the compiler performs NO optimizations on instructions that use the data
    - All reads and writes are executed even if they seem redundant
  - Force a memory read each time the variable is used

```
int flag = 500;
foo() {
. . .
while (flag != 10) {...;}
Raise_alarm();
. . .
}
```

Memory

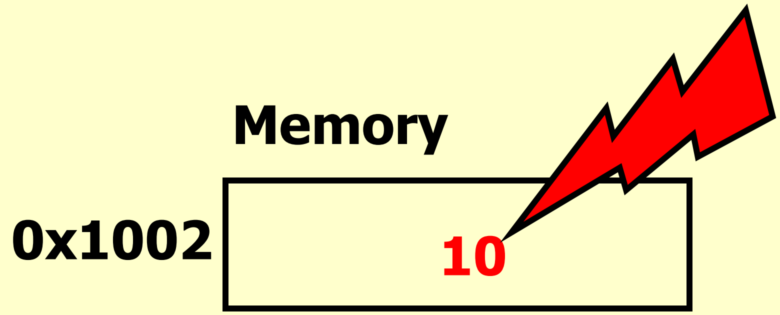
0x1002 =&flag: 500

```
        LDIU    1002h, AR4
        . . .
L11     . . .
        CMPI    10h, *AR4
        BEQ             L11
        CALL    &Raise_alarm
        . . .
```

OPTIMIZE →

```
        LDIU    1002h, AR4
        . . .
        LDIU    *AR4, R7
L11     . . .
        CMPI    10h, R7
        BEQ             L11
        CALL    &Raise_alarm
        . . .
```

```
int flag = 500;
foo() {
...
while (flag != 10) {...;}
Raise_alarm();
...
}
```

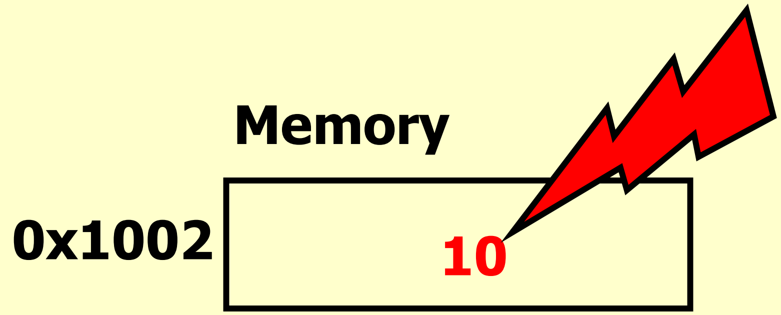



```
        LDIU    1002h, AR4
        ...
L11     ...
        CMPI    10h, *AR4
        BEQ             L11
        CALL    &Raise_alarm
        ...
```

OPTIMIZE

```
        LDIU    1002h, AR4
        ...
L11     LDIU    *AR4, R7
        ...
        CMPI    10h, R7
        BEQ             L11
        CALL    &Raise_alarm
        ...
```

```
volatile int flag = 500;
foo() {
. . .
while (flag != 10) {...;}
Raise_alarm();
. . .
}
```




```
        LDIU   1002h, AR4
        . . .
L11     . . .
        CMPI   10h, *AR4
        BEQ           L11
        CALL   &Raise_alarm
        . . .
```

```
        LDIU   1002h, AR4
        . . .
        LDIU   *AR4, R7
L11     . . .
        CMPI   10h, R7
        BEQ           L11
        CALL   &Raise_alarm
        . . .
```

‘volatile’ Keyword

- Many compilers support the use of the keyword **volatile**
  - Identifies variables as those that are shared
  - Tells the compiler to NOT store the variable value in a register
  - Ensures that the value is the most recent
    - There is no problem of **inconsistency** between a register and memory location that both refer to the same variable
  - Reads and Writes to volatile variables are ordered and the order is retained in the compiled program flow
- Upshot: volatile forces the compiler to use a sequential direct to memory map model
  - Similar to forcing a “write-through” cache
  - <u>Number</u> and <u>order</u> of original reads and writes specified in the code is conserved

Interrupts, Traps, and Exceptions

- Interrupt - a raised CPU signal
  - External - raised by an external event
  - Internal - generated by on-chip peripherals
  - Can happen any time -- **asynchronous interrupts**
- Trap - software interrupt (**synchronous interrupts**)
  - Mechanism that changes control flow
  - **Bridge** to supervisor mode (system calls)
  - Requires additional data to be communicated
    - Parameters, type of request
    - Return values (status)
- Exception - unprogrammed control transfer
  - General term for synchronous interrupts
  - On some machines these include internal async interrupts

Interrupts, Traps, and Exceptions

- External Interrupts - a raised CPU signal
  - Asynchronous to program execution
  - May be handled between instructions
  - Simply suspend and resume user program
- Traps (and exceptions)
  - Exceptional conditions (overflow)
  - Invalid instruction
  - Faults (non-resident page in memory)
  - Internal hardware error
  - Synchronous to program execution
  - Condition must be remedied by the handler

Restarting After a Synchronous Interrupt

- Restart - start the instruction over
  - May require many instructions to be restarted to ensure that the state of the machine is as it was
- Continuation - set up the processor and start at the point (midstream) that the fault occurred
  - Process state is so complete that restart is not necessary
  - A second processor handles the interrupt so the first processor can simply continue where it left off

Traps - Implementing System Calls

- Operating System
  - Special program that runs in **priviledged** mode and has **access to all of the resources and devices**
  - Manages and **protects** user programs as well as resources
    - Via keeping user programs from directly accessing resources
    - Uses a separate **user** mode for all non-OS related activities
  - Presents **virtual** resources to each user
    - Abstractions of resources are easier to use
      - **files vs. disk sectors**
      - **Virtual memory vs physical memory**
- Traps are OS requests by user-space programs for service (access to resources)
  - Begins at the handler

MicroBlaze Interrupt Handlers

- MicroBlaze uses the GNU conventions which specify a function attribute for a defined interrupt handler:

```
void function_handler () __attribute__ {{interrupt_handler}};
```

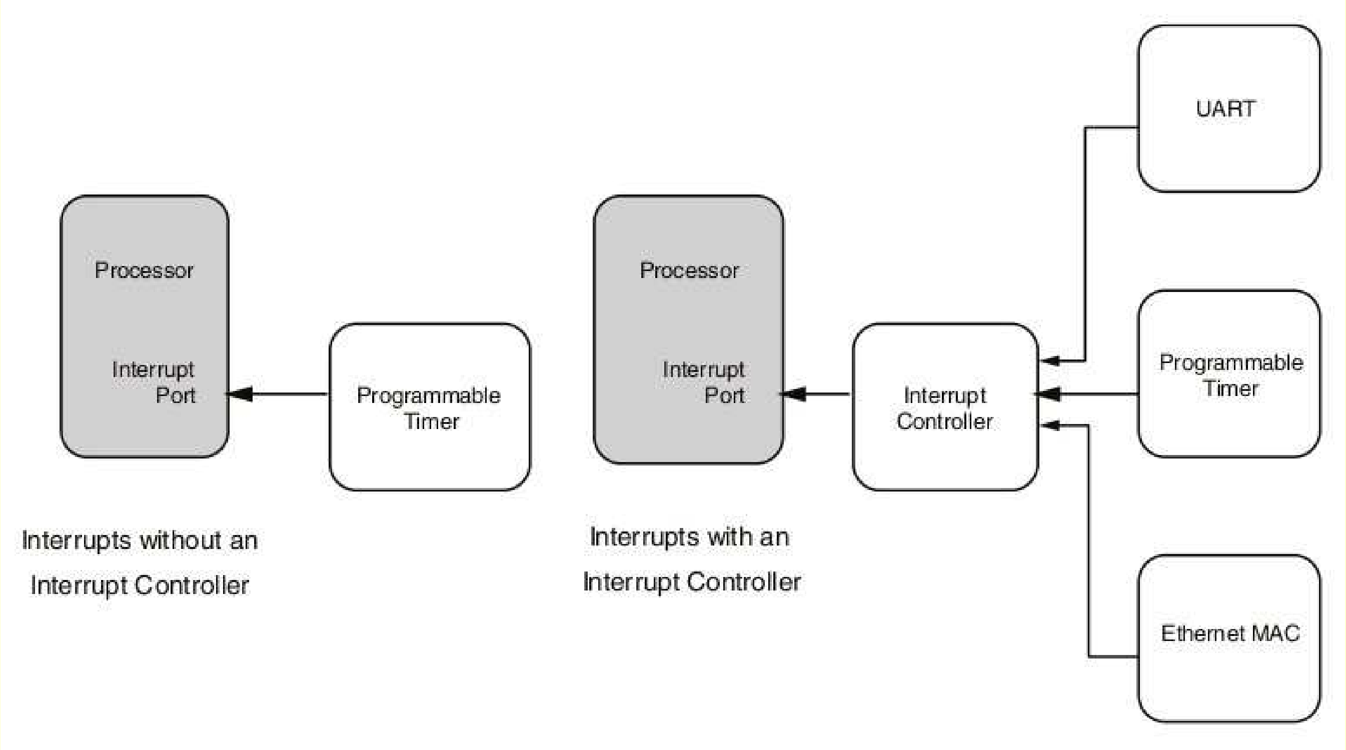
  - Note: attribute is applied to the function prototype – not it definition!!
  - Interrupts might call subroutines – they need to be volatile-safe this can be done:

```
void function_subhandler () __attribute__ {{save_volatiles}};
```

Attributes	Functions
interrupt_handler	This attribute saves the machine status register and all the volatiles, in addition to the non-volatile registers. `rtid` returns from the interrupt handler. If the interrupt handler function is a leaf function, only those volatiles which are used by the function are saved.
save_volatiles	This attribute is similar to `interrupt_handler`, but it uses `rtsd` to return to the interrupted function, instead of `rtid`.

MicroBlaze Interrupts

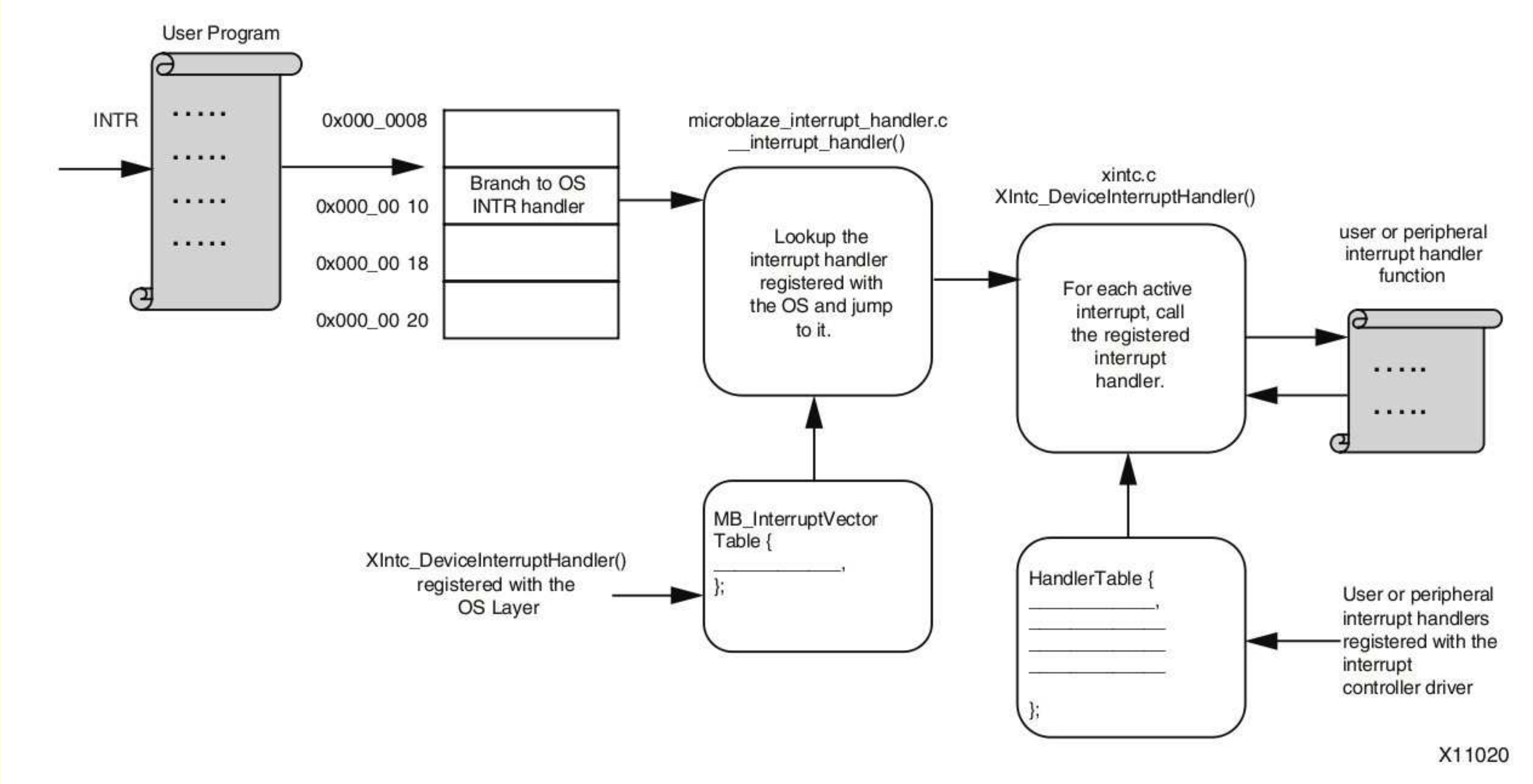
- Only a single bit of interrupt for MB
  - Need a controller to manage multiple sources
  - Fortunately controllers are part of the library



MB Interrupt flow

- Enable Interrupts in MSR
- Hardware disables Interrupts
- Handler saves registers onto stack; saves return from
  Vector dispatch table
- Transfer to user handler
  Interrupt controller is managed by this handler – typically vectored dispatch to specific interrupt handler for given source
- Device-specific handler manages device then returns
  - Stack is unwound by successive returns to final return from interrupt– which re-enables interrupts

MB Interrupt Flow



Interrupt Conclusions

- Interrupts allow possibility of preemption of tasks
  - Add greatly to complexity of programming
- Enable designers to split work - modularity
  - Background work (tasks performed while waiting for interrupts)
    - Unaware of foreground work
  - Foreground work (tasks that respond to interrupts)
- Very strong reasons to minimize time spent in handlers
- They can be
  - **Precise** – programmed system response to known event
  - **Imprecise** – reset or abort behavior to known state; usually fault or catastrophic event
  - **Asynchronous** – obeying only physical limits of timing

Reactive System Reprise

- Game Plan: Spend absolute minimum time in interrupt handlers
- Treat program flow as extended finite automata
  - Computation parts broken into fragments that must be done given current state and current events
  - Control-flow follows next state transition (event and state dependent), often a bit of prologue and epilogue code to ensure state independence.
  - Interrupt handler triggers updates to FSM state
- Use vector dispatch (I.e. function pointers or computed go-to) to minimize transition time
- Upshot: worst case response time could be much lower
  - Never allow handler long run-time for any event
- Issue: requires alternative view of program organization…