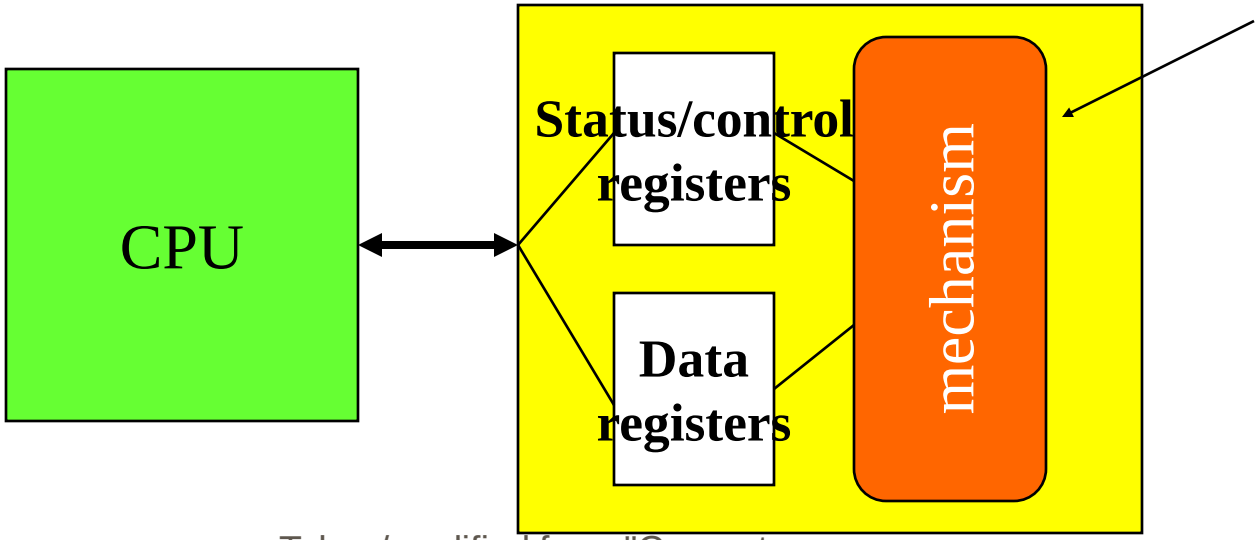
I/O, Interrupts, CPUs

- Input and output mechanisms
- Interrupts
- CPU performance: Pipelining
- CPU power consumption

I/O devices

- Usually includes some non-digital components.
- Typical digital interface to CPU:

Programmable device: e.g., A/D converter bank, serial port, …



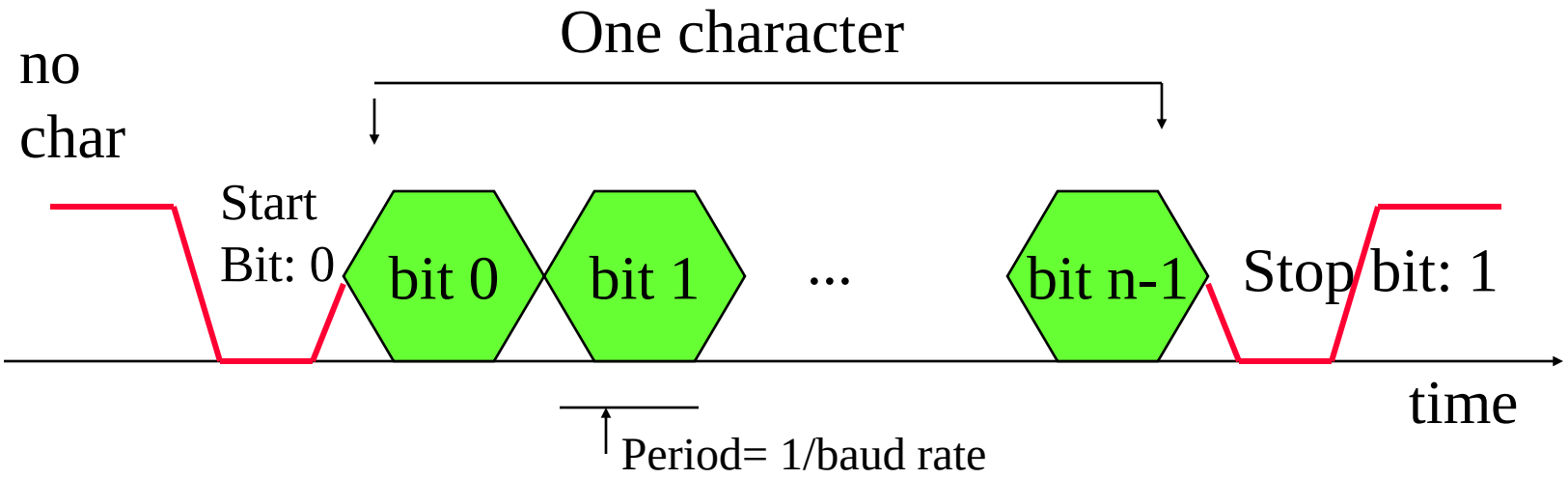
Taken/modified from "Computers as Components, W. Wolf"

Application: 8251 UART

- Universal asynchronous receiver transmitter (UART): provides serial communication, e.g., serial port connection on PC's
- 8251 functions are integrated into a standard PC interface chip.
- Allows many communication parameters to be programmed.

Serial communication

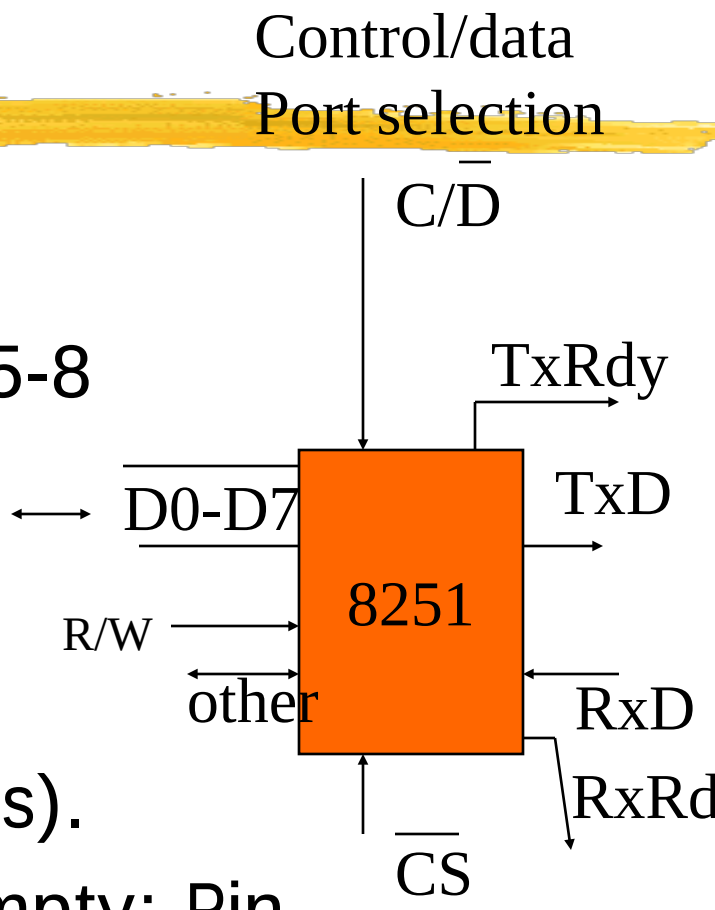
- Characters are transmitted separately:



Taken/modified from "Computers as Components, W. Wolf"

Serial communication parameters

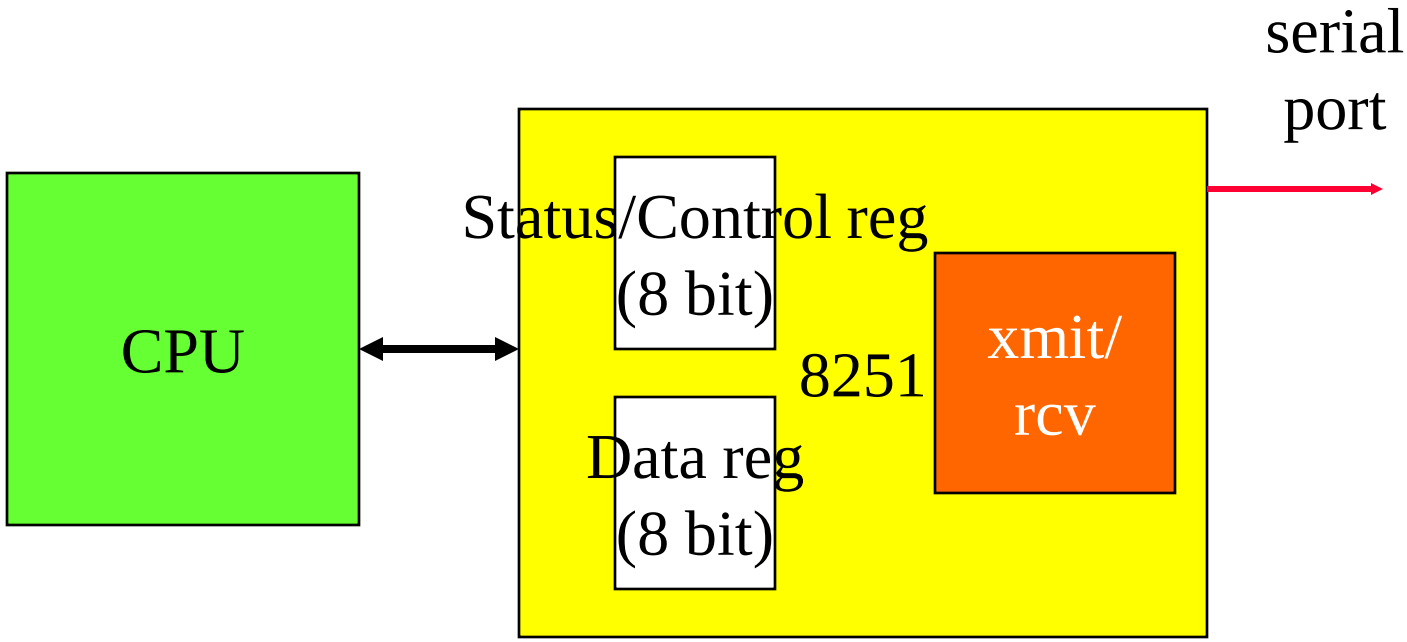
- Baud (bit) rate.
- Number of bits per character: 5-8
- Parity/no parity.
- Even/odd parity.



- Length of stop bit (1, 1.5, 2 bits).
- Transmitter/Receiver Ready/Empty: Pin indicating status of Receiver/Transmitter

Taken/modified from "Computers as Components, W. Wolf"

8251 CPU interface



Taken/modified from "Computers as Components, W. Wolf"

Programming I/O

- **Two ways that microprocessors can support I/O:**
  - **special-purpose I/O instructions;**
  - **memory-mapped load/store instructions.**
- **Intel x86 provides `in`, `out` instructions.**

  Most other CPUs use memory-mapped I/O.
- **I/O instructions do not preclude memory-mapped I/O, i.e., can use memory map as well.**

ARM memory-mapped I/O

- Define location for device:

```
DEV1 EQU 0x1000
```

Pseudo-op to define DEV1

- Read/write code:

```
LDR r1,#DEV1 ; set up device adrs
LDR r0,[r1] ; read DEV1
```

```
LDR r0,#8 ; set up value to write
STR r0,[r1] ; write 8 to device
```

Peek and poke:

Use pointers to manipulate I/O

- High level language interface:

```
char peek(char *location) {
        return *location;
}
```

-
```
char *devloc= 0x1000;
char dev_status;
dev_status = peek(devloc); //read
  device
```

… Peek and poke

```
void poke(char *location, char
  newval) {
        (*location) = newval;
  }
```

```
EXAMPLE:  poke (DEV1, 8);
    //write 8 to device register
```

Busy/wait input/output

- Devices are typically slower than the CPU:
  e.g., if we start writing characters before the device has finished with the first one, there may be a problem ...

- Asking I/O if it has finished by reading a bit in the status register:
  - **Busy waiting on a status bit**
  - **Periodically polling the status bit (with no waiting)**

- Example: EOC line of an A/D converter

Taken/modified from "Computers as Components, W. Wolf"

Example: Write a sequence of characters to an output device → device has status/character registers

```
#define OUT_CHAR  0x1000  //character register address
#define OUT_STATUS 0x1001 //status register address

char *mystring=``Hello'';  //pointer to a char string
char *current_char; /* pointer to  current
                       position */
current_char = mystring;

while (*current_char != '\0') {
   poke(OUT_CHAR,*current_char);
   while (peek(OUT_STATUS)!= 0); //1=busy
   current_char++;
}
```

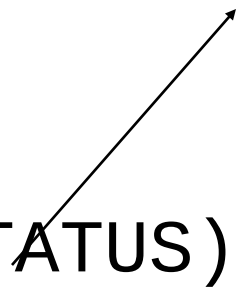
Simultaneous busy/wait input and output

- **Repeatedly read a character from an input device and write it to an output device.**

```
#define  IN_DATA      0x1000     //define addresses
#define  IN_STATUS    0x1001
#define OUT_DATA      0x1102
#define OUT_STAT      0x1103
```

```
while (TRUE) {
  /* read */
  while (peek(IN_STATUS) == 0) ;
  achar = (char) peek(IN_DATA);
```

Input device sets *IN_STATUS=1 when a new char arrives



Taken/modified from "Computers as Components, W. Wolf"

... Simultaneous busy/wait input and output

```
/* write */
poke(OUT_DATA,achar);
poke(OUT_STATUS,1); /* turn on
                       device */
/* wait until done */
while (peek(OUT_STATUS) != 0);
} /* end of while(TRUE) */
```

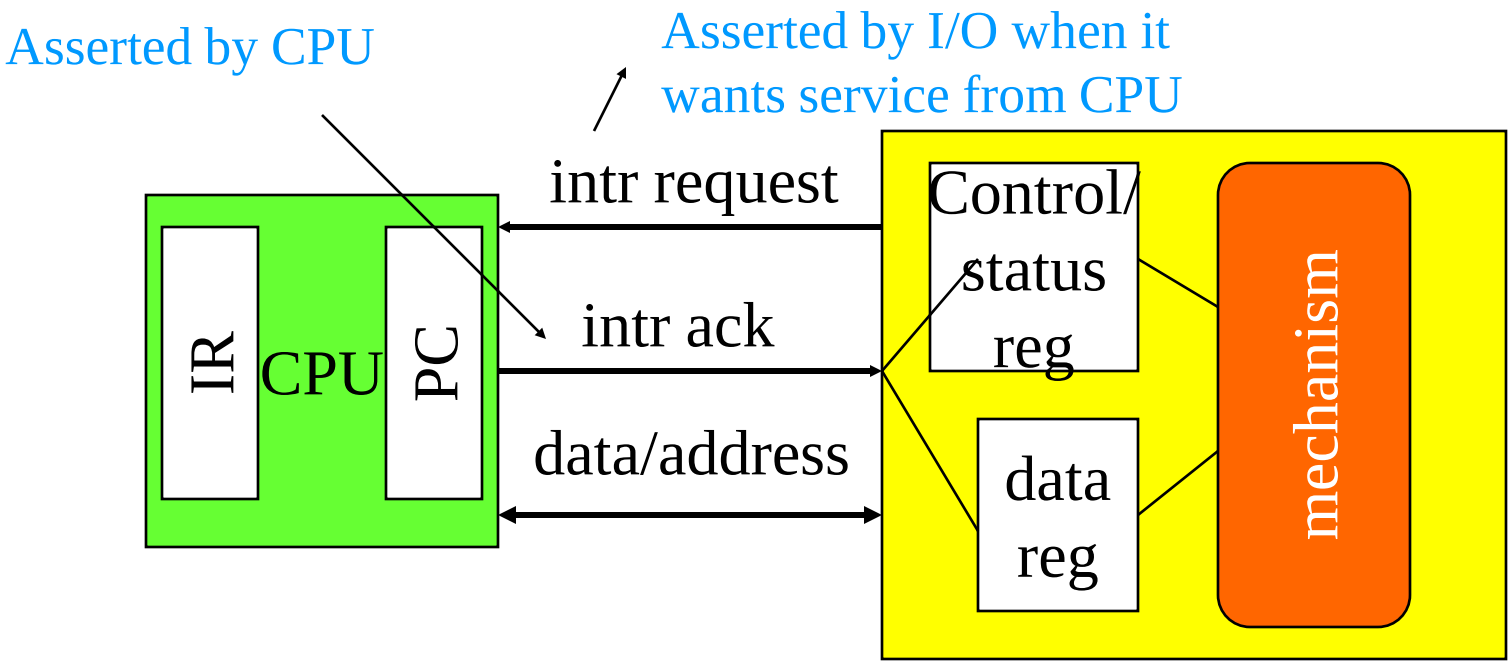
Taken/modified from "Computers as Components, W. Wolf"

Interrupt I/O

- Busy/wait is very inefficient.
  - CPU can't do other work while testing device, e.g., can control other I/O or do computation
  - Hard to do simultaneous I/O.
- Alternatives: Polling with no busy waiting, interrupts
  - Interrupts allow a device to change the flow of control in the CPU.
  - Similar to subroutine call to handle a device.

Taken/modified from "Computers as Components, W. Wolf"

Interrupt interface



Taken/modified from "Computers as Components, W. Wolf"

Interrupt sequence

- Device asserts interrupt line.
- CPU acknowledges request.
- CPU calls handler (could be vectored interrupt).
- Software processes request.
- CPU restores state to foreground program.

Interrupt behavior

- Based on subroutine call mechanism.
- Interrupt forces next instruction to be a subroutine call to a predetermined location.
  - Return address is saved to resume executing foreground program (the program that runs when no interrupt is being handled)

Interrupt physical interface

- CPU and device are connected by CPU bus (Data/Address/Control).

- CPU and device handshake:
  - device asserts interrupt request;
  - CPU asserts interrupt acknowledge when it can handle the interrupt.

Example: Character I/O handler

→Repeatedly read char from input and write it to output

```
char achar;   /*for passing to main program */
char  gotchar;    /* signaling a new character */

// Interrupt routine
void input_handler() {
  achar = peek(IN_DATA);    /* get char */
  gotchar = TRUE;           /* signal to main() */
  poke(IN_STATUS,0); /* reset status for next
                        transfer */
}
```

Taken/modified from "Computers as Components, W. Wolf"

Example: Interrupt-driven main program

```
main() {
  while (TRUE) {
     if (gotchar) {
          poke(OUT_DATA,achar);
          gotchar = FALSE;
          }
     }
  }
```

Problems with interrupts

- Race conditions

```
int   position, velocity;
void f1(){
        position = peek(POS_DEV_ADDRESS);
        velocity = peek(VEL_DEV_ADDRESS);
    }
void f2(){
        while(1) {
                plot(position*POS_SCALE_FACTOR);
                plot(velocity*VEL_SCALE_FAC);
                }
    }
```

This code is run by a periodic interrupt line (ISR)

Running as a main program

… interrupts

- What if interrupts are disabled in f2( )?

  ➔ increases interrupt latency

- Interrupt vs polling: **Interrupt latency (ex. 60 micro-sec), polling a register (ex. 6 micro-sec)**
  - FAST DEVICES: poll
  - SLOW DEVICES: interrupt
  - If interrupt latency is high, convert periodic interrupts to periodic tasks

  - Example: Let Tpoll=6us (polling overhead), Tint= 60us (int latency), T=1000us (average periodic occurrence of an event)

    ➔ THEN: T_poll/(T/N) =0.03 if polled N=5 times during T

    vs T_int/T=0.06 (utilizes CPU more)

Taken/modified from "Computers as Components, W. Wolf"

Debugging interrupt code-
in assembly programming

- What if you forget to restore registers?
  - Program can exhibit mysterious bugs, e.g., if a register is not properly restored in the interrupt handler
  - Bugs will be hard to repeat---depend on interrupt timing.

Sources of interrupt overhead

- Handler execution time.
- Interrupt mechanism overhead: branch penalty
- Register save/restore.
- Penalties related to the specific architecture: pipeline/cache ...
- Coding the interrupt handler in assembly may resolve some of the issues, e.g., more efficient use of registers

Next → CPUs

- CPU performance:

  How fast it can execute instructions→ increasing throughput by pipelining

- CPU power consumption

Elements of CPU performance

- Cycle time: How fast CPU executes an instruction
- CPU pipeline: Modern CPUs are pipelined machines
- Memory system: Can affect overall performance.

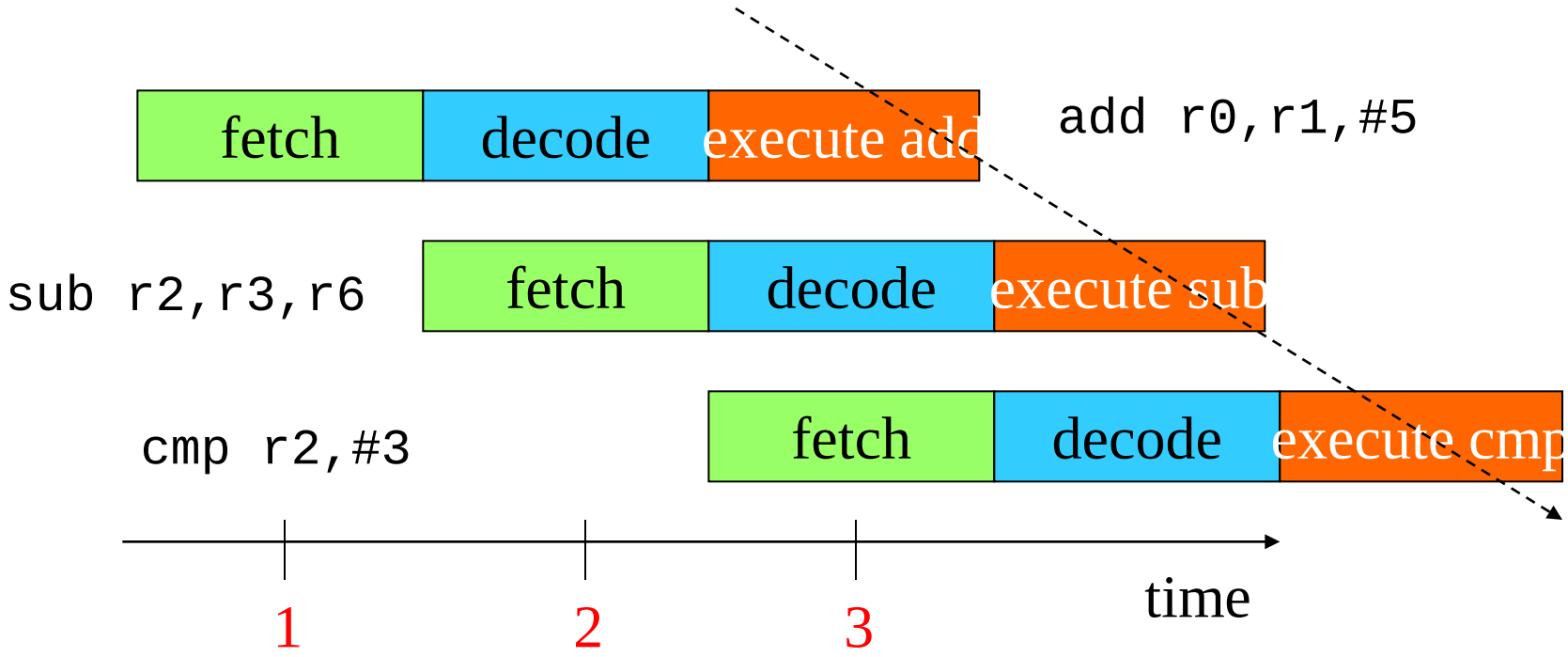
Pipelining

- Several instructions are executed simultaneously at different stages of completion.
- Various conditions can cause pipeline bubbles that reduce utilization:
  - branches;
  - memory system delays;

Pipeline structures

- Both ARM and SHARC have 3-stage pipes:
  - fetch instruction from memory;
  - decode opcode and operands;
  - execute.
- Without pipeline we need at least 3 cycles to complete an instruction
- With pipeline 1 cycle (on average)

ARM pipeline execution



Taken/modified from "Computers as Components, W. Wolf"

Performance measures

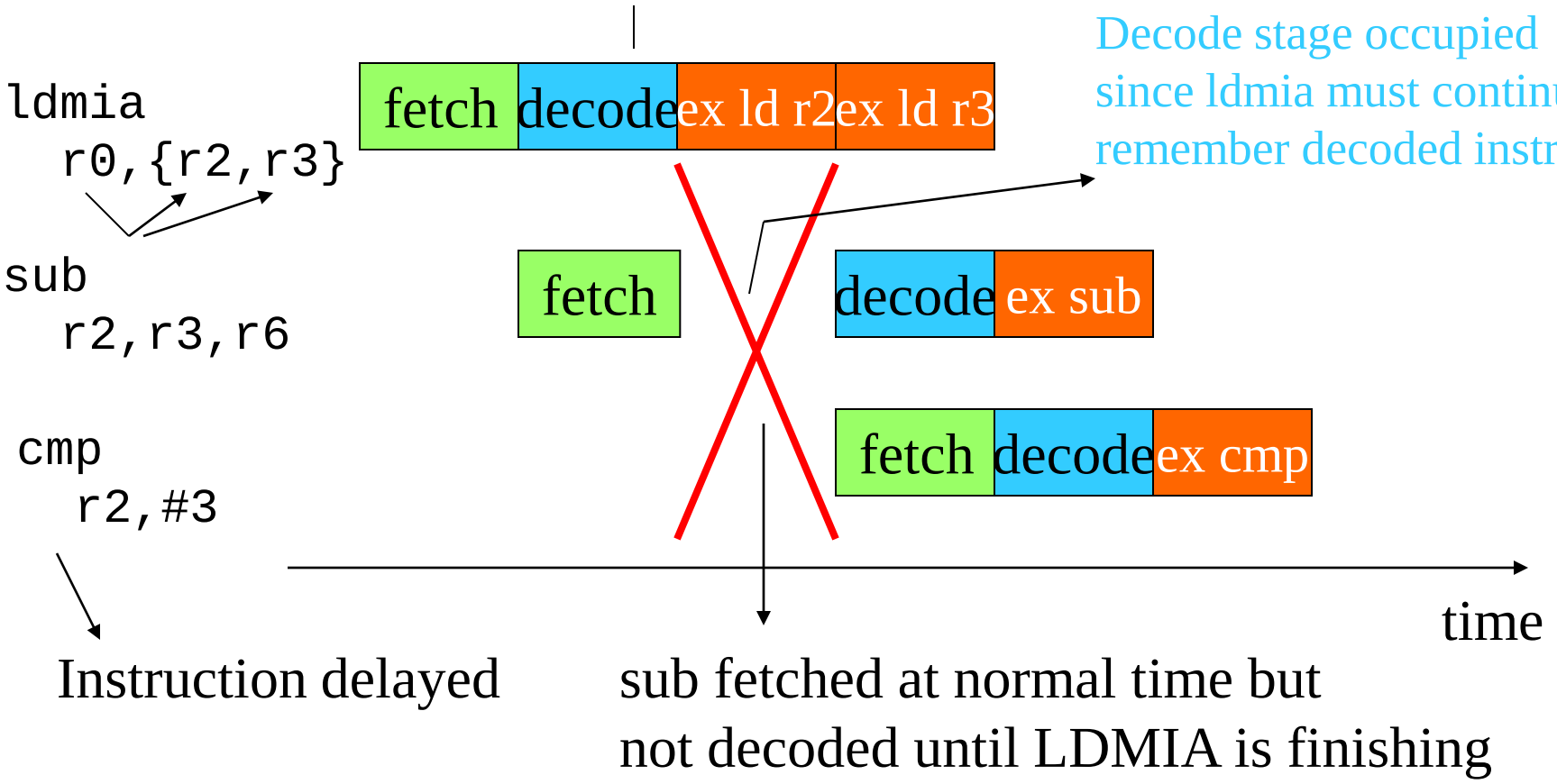
- Latency: time it takes for an instruction to get through the pipeline: 3 clock cycles
- Throughput: number of instructions executed per time period: 1/cycle
- Pipelining increases throughput without reducing latency.

Taken/modified from "Computers as Components, W. Wolf"

Pipeline stalls: Instructions too complex to complete in one cycle

- If every step cannot be completed in the same amount of time, pipeline stalls.
- Bubbles introduced by stall increase latency, reduce throughput.

ARM multi-cycle LDMIA (load multiple) instruction

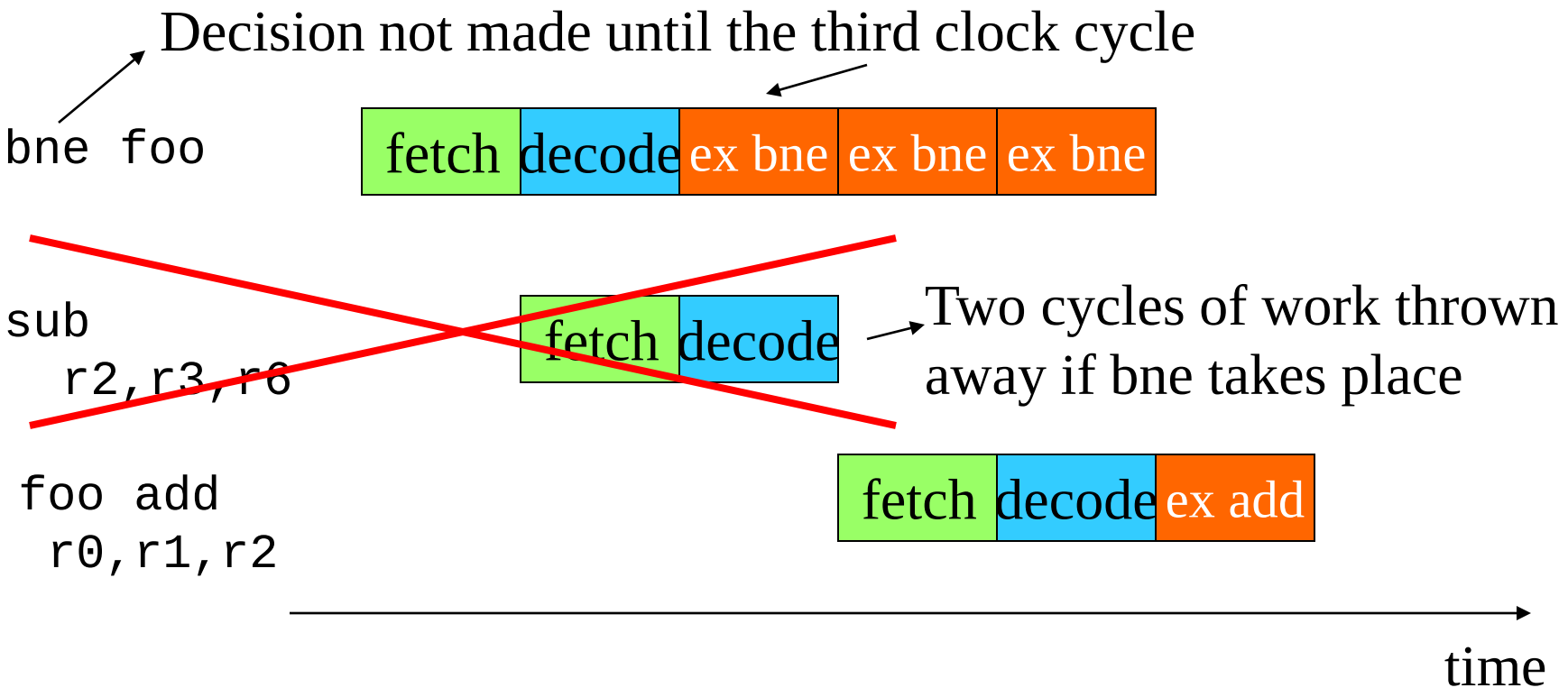


Taken/modified from "Computers as Components, W. Wolf"

Control stalls: due to branches

- Branches often introduce stalls (branch penalty).
  - Stall time may depend on whether branch is taken.
- May have to squash instructions that already started executing.
- Don’t know what to fetch until condition is evaluated.

ARM pipelined branch



CPU power consumption

- Most modern CPUs are designed with power consumption in mind to some degree.

- Power vs. energy:
  - heat depends on power consumption;
  - battery life depends on energy consumption.

CMOS power consumption

- Voltage drops: power consumption proportional to V^2.
- $P = \frac{1}{2} f C V^2$ (CMOS Inverter circuit)
- Toggling: more activity means more power → Reducing speed reduces power
- Leakage: basic circuit characteristics; can be eliminated by disconnecting power.

Taken/modified from "Computers as Components, W. Wolf"

CPU power-saving strategies

- Reduce power supply voltage.
- Run at lower clock frequency.
- Disable function units with control signals when not in use.
- Disconnect parts from power supply when not in use.

Power management styles

- Static:
  - E.g., User-activated power-down mode. Entered by an instruction.
- Dynamic, based on CPU activity:
  - Example: disabling off function units, e.g., certain CPU sections when instructions do not need them

Power-down costs

- Going into a power-down mode costs:
  - time;
  - energy.
- Must determine if going into mode is worthwhile → Initialization may take time and energy
- Can model CPU power states with power state machine.

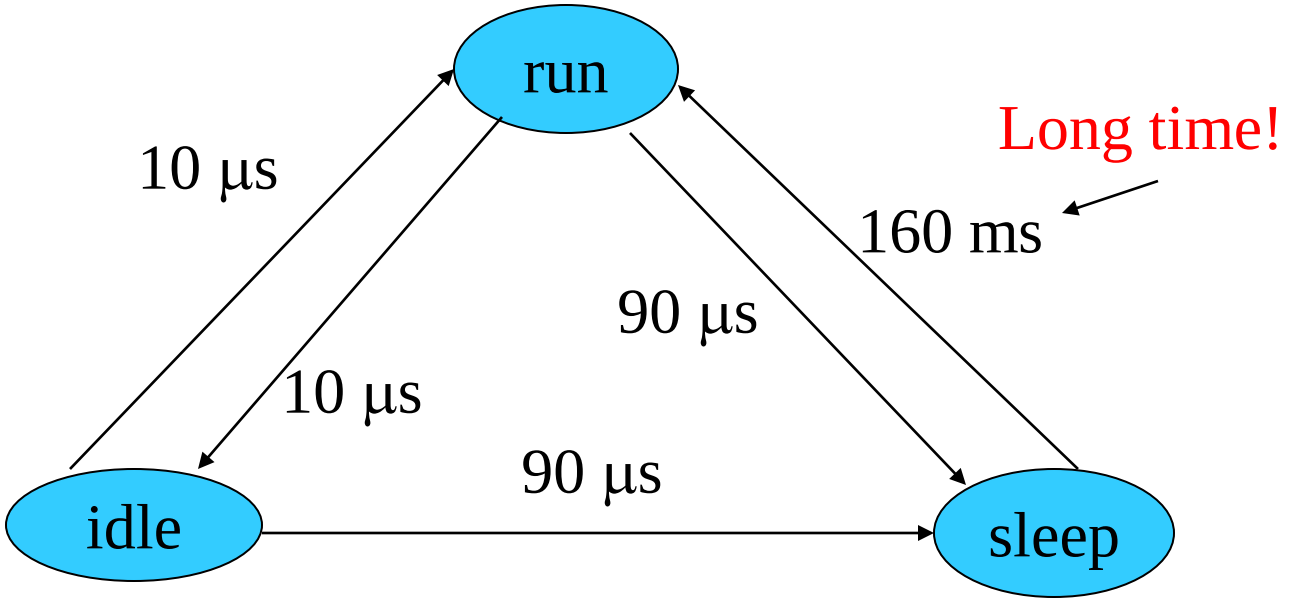
Application: StrongARM SA-1100 power saving

- **Processor takes two supplies:**
  - **VDD is main 3.3V supply → powers the CPU core**
  - **VDDX is 1.5V → other logic, e.g., power manager**
- **Three power modes:**
  - **Run: normal operation.**
  - **Idle: stops CPU clock, with logic still powered, e.g., clock, o/s timers, general purpose IO**
  - **Sleep: shuts off most of chip activity; 3 steps, each about 30 μs; wakeup takes > 10 ms.**

Taken/modified from "Computers as Components, W. Wolf"

SA-1100 Power States

P_{run} = 400 mW



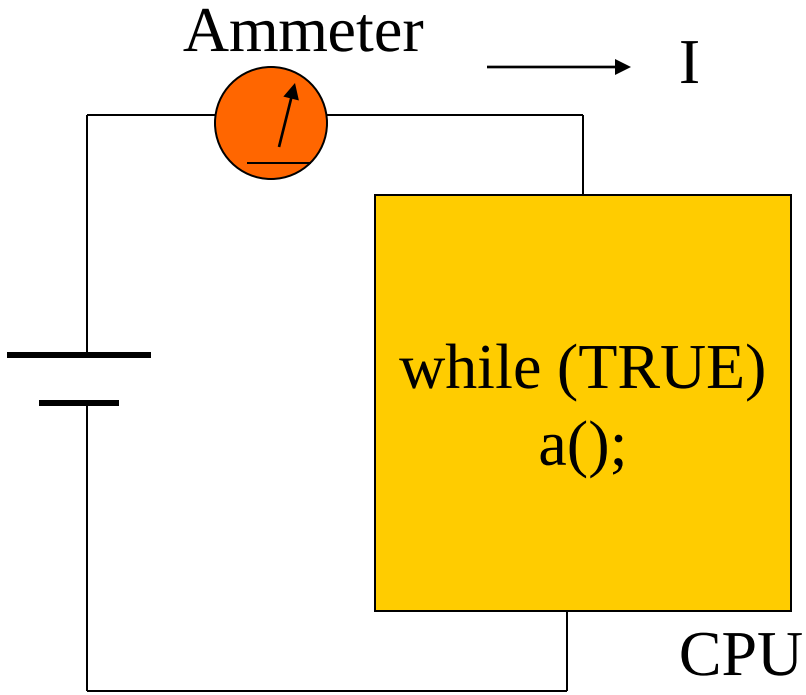
P_{idle} = 50 mW

P_{sleep} = 0.16 mW

Taken/modified from "Computers as Components, W. Wolf"

Measuring energy consumption

- Execute a small loop, measure current:



Calculate power:

Power with a() in the while loop –

Power without a() in the while loop

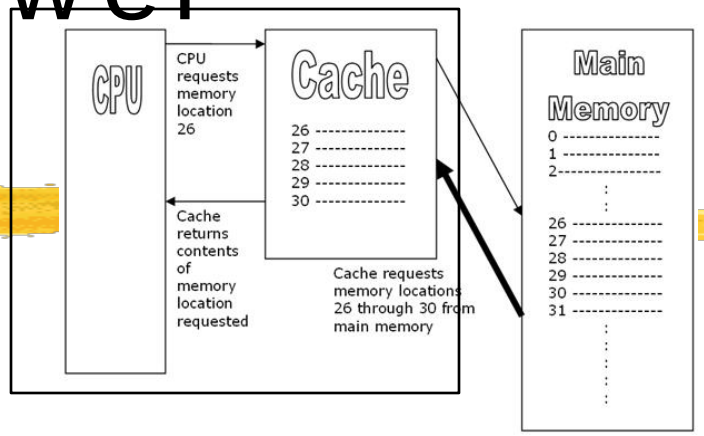
Taken/modified from "Computers as Components, W. Wolf"

Sources of energy consumption

- Relative energy per operation:
  - **memory transfer (DRAM): 33 → most expensive**
  - **external I/O: 10**
  - **SRAM write: 9**
  - **SRAM read: 4.4**
  - **multiply: 3.6**
  - **add: 1**
  - **register access (<1) → most efficient**

Taken/modified from "Computers as Components, W. Wolf"

Cache Memory Power Consumption



- Cache (on CPU chip) is
- power hungry → **built from SRAM**
- Energy consumption has a sweet spot as cache size changes:
  - **cache too small: program thrashes, burning energy on external memory accesses;**
  - **cache too large: cache itself burns too much power.**
  - ➔ **There is an optimum cache size**

Taken/modified from "Computers as Components, W. Wolf"

Assignment 3

- Q3-1, Q3-4 (Also, do the same question for HCS12 by writing a C program and observing the Assembly Code), Q3-8, Q3-31,