CSC230 Notes after Midterm

Stack and Function Calls

- Last in First Out storage structure
- AVR includes stack pointer register (SP) in the I/O register area
- Stack required for interrupts and procedure calls
- Used for temporary storage and to free a register for another use
- Used for local variable storage and storage for parameters
- Stack is used when interrupt handler is invoked
- Stack is in SRAM memory
- SP keeps track **top of the stack, holding address where the next byte would be placed**
- Stack is implemented from higher memory locations to lower memory locations
    - Stack **PUSH** command **decreases SP**
    - Stack **POP** command will first increment SP (move pointer from next byte to the current byte)
    - SP never points to data actually on the stack because it is decremented when info is pushed onto stack
- Bottom of the stack = end of SRAM at address $21FF
- SP implemented as **two** 8-bit registers in I/O space
    - Register address 0x3D = SPL
    - Register address 0x3E = SPH
- If stack is **not** empty, SP+1 is top of stack
    - SP is pointing to the **NEXT** byte in the stack
- **There is no check for stack overflow**

- Functions, procedure, subroutine interchangeably
- Functions returns a value, procedure or subroutine **does NOT**
- Function entry point with a label and exit with RET instruction
    - Make sure the function cannot be executed without being called
- Use call instruction with function's address to use function from any point
- **When call instruction executed, the processor will push the return address onto the stack, then jump to the function**
    - **When RET instruction is reached, the PC will pop that address from the stack which will resume back to where we exec the call instruction**
- Can use registers, stack or memory to return values
    - Have the function place return value in a register
    - Return value on stack – pop return **address** from stack, push return **value,** push return **address** again
- Arguments belong to calling function, parameters belong to the called function
    - Call by value and call by reference are a way of passing arguments to a function
    - Call by value – argument is copied into the parameter allowing function to use the value (see example on slides 32)

    - Call by reference – caller makes the address of the argument available to the function (see example on slides 32)
- Inline parameters allocated in the middle of the exec code
    - Call instruction in program followed by data allocation
- Local storage – used only while function is executing
    - Should be allocated as function is called and deallocated as the function completes its task
    - If local storage exceeds available space in registers, it must allocate storage from SRAM
- Stack Frame – proportion of the stack used by a function during its lifetime
    - Contains parameters, return address, local storage for function

Interrupts

Slides 1-22

- Interrupt system of a computer allow programs to respond immediately to external events
- Interrupt is a signal that interrupts the flow of the currently running program
- Can be triggered by external or internal signals
- Caused by software or hardware
- Interrupt service routine (ISR) when the program gets interrupted
    - Process of interrupting is called servicing the interrupt
- CPU jumps to a pre-defined location when interrupt occurs
    - Pre-defined location called **interrupt vector address**
- Once CPU completes routine, it continues from where it made the jump
- For AVR, the PC register is the only register that gets saved
    - It is up to the user to save other registers
    - Internal interrupts – generated by on-chip I/O devices and **don't have individual enable/disable bit**
    - External interrupts – generated by external I/O devices and **has individual enable/disable bit**
    - I-bit in SREG is a global interrupt enable/disable
    - Each interrupt has 4-byte interrupt vector, a vector number, an integer from 1 to n, and the max number of interrupts
        - All interrupt vectors are stored in the flash memory
    - Priority of each interrupt is determined by its vector number (low = high priority)
- Conventional polling – CPU looks at status changes of the I/O device
- Vector_address = (k-1)*2
    - This address loaded into PC, cause a branch to interrupt vector, branch to ISR
- Reset signal interrupt = exec begins at address $0000
    - Reset is a Non-maskable interrupt (NMI)
- Global Interrupt flag
    - Located in SREG **bit 7**
    - Flag is cleared when a reset occurs
    - Interrupts are enabled by setting the interrupt flag
    - SEI instruction easiest to set flag

    - CLI clears the flag
    - If the I-bit is enabled in the ISR, nested interrupts are **allowed**
    - I-bit is cleared after an interrupt has occurred with the RETI instruction
    - **Interrupt handlers must save the SREG contents by pushing the SREG contents onto a stack and popping contents back after interrupt**
- Interrupt vector contains a branch instruction to braches to the first instruction of the ISR
- RETI to return from interrupt
- AVR External Interrupts
    - External interrupts can be triggered using 2 sets of pins: INTn pins and PCINTn pins
        - The eight INTn (0-7) have dedicated interrupt vector address
        - PCINTn share the **same** interrupt vector address
            - PCINTn (0-7) will trigger interrupt vector address PCINT0 = $0012
            - PCINTn (8-15) will trigger interrupt vector address PCINT1 = $0014
            - PCINTn (16-23) will trigger interrupt vector address PCINT2 = $0016
    - Interrupt vectors are simply jump instructions that transfers to an interrupt routine
    - Interrupt vectors reside in program memory at $0000
    - Asynchronous – does not require I/O clock
        - INT0 – INT3 (External Interrupt Control register A **EICRA**)
        - All PCINT
    - Synchronous – requires I/O clock
        - INT4 – INT7 (External Interrupt Control register B **EICRB**)
    - External interrupt mask register (EIMSK) – enable or disable an interrupt by setting 1 or 0

*********END********

- Power-on reset – MCU is reset when supply voltage is below Power-on reset threshold
- External Reset – MCU is reset when low level is present on RESET pin
- Watchdog reset – MCU is reset when watchdog timer expires
- Watchdog Timer
    - Used to detect software crash
    - 8 different periods determined by WDP2, WDP1, WDP0 bits in WDTCR
    - If enabled, generates watchdog reset interrupt when period expires, so program needs to reset it before its period expires by exec instruction WDR
- Timer Interrupt
    - Round Robin scheduling – all tasks take turn to exec
    - Real time scheduling – some tasks must be started at a certain time and finish by deadline
    - Used to implement clock
    - Used to synchronize tasks – start Task A iff Task B completed over a certain time
- AVR timers
    - Two 8 bit and four 16 bit timer counters
        - Counter mode – always counts up, creates overflow
        - Compare mode – compares counter to a set of registers and interrupts if values equal

    - Counter 0
        - 8 bit
        - Wrap around up counter
        - Interrupt on overflow
    - Counter 1
        - 16 bit
        - Compare mode
        - Up counter
        - Interrupt on overflow, compare A/B
        - Can act as 8 – 10 bit up-down counter
- Timer 0
    - Control register (TCCR0)
    - Counter0 (TCNT0)
- Timer 1
    - Control register A and B (TCCR1A/B)
    - Input capture register (ICR1)
    - Counter1 output compare register A and B (OCR1A/B)
    - Counter1 (TCNT1)

C Programming

-

Performance

Slides 13-40

- Execution time = OS time + program time
- Real time
    - Counts everything, disk memory access, I/O
- CPU time
    - Time spent executing lines of code in our program
    - Does not count I/O or time spent running other programs
    - Broken up into **system** time and **user** time
- Response time (latency)
    - How long does it take to run?
    - Time to execute a job
- Throughput (rate of work)
    - How many jobs can the machine run at once?
    - Average execution rate
    - Amount of work getting done
    - Number of processes executed in a unit of time
- Bus
    - Synchronous (memory)
    - Asynchronous (peripherals)
    - Control lines

    - Address lines
    - Data lines

- Processor Clock
    - Clock is a timing signal
    - **Clock cycles** = each length of period P time for 1 clock cycle
        - P = sec / cycles
    - **Clock rate** R = 1 / P
        - R = cycles / sec
    - Machine instructions divided into **Cycles per second (Hertz)**
    - 500 million cycles per **second** = 500MHz
- Performance Equation
    - T = performance time
    - N = actual number of executed instructions (include number of times a loop executed)
    - S = number of basic 1-clock steps needed for one instruction
    - R = clock rate (cycles per second)
    - **T = (N * S) / R**
        - **Reduce T by reducing Numerator or increasing Denominator**
- SPEC Ratings (System Performance Evaluation Corporation)
    - **SPEC rating = running time on a reference computer / running time on computer under test**
- **Performance = 1 / Execution time**
- CPU Performance and clock
    - **CPU Exec time for a program = CPU clock cycles for a program / Clock rate**
    - **CPU Exec time for a program = CPU clock cycles for a program * Clock cycle time**
- CPI – Clock cycles per instruction
    - **CPI = CPU clock cycles / Instruction count**
    - **CPU clock cycles = Instruction count * CPI**
- CPU Performance Equation
    - **CPU time = Instruction count * CPI / Clock rate**
    - **CPU time = Instruction count * CPI * P**

Caches

slides 1-42, 58-66

- Memory hierarchy consists of many levels of memory with different speeds and sizes to give the illusion of the largest level of hierarchy but accessed from the fastest memory
- Faster memory, cache, ram closer to CPU and slower, less expensive further
- RAM (Dynamic memory DRAM) = Main memory = Memory | Second level Cache (Static memory SRAM)
- All data stored at the lowest level
- Data is copied to cache and RAM when required
- Cache memory is where data and instructions go when they are being used **very very soon**
- **Principle of Locality**

    - Programs access a small portion of the address space at any time
    - Temporal Locality – Keep most recently access data closer to processor
    - Spatial Locality – Move items related to the accessed item close by
- Hit – data appears in a block in upper level (cache)
    - Hit rate – fraction of memory access found in the upper level
    - Hit time – time to access the upper level: access time + determine hit/miss time
- Miss – data not found in upper level (cache)
    - Miss rate – (1-(hit rate)) – fraction of memory access **not** found in the upper level
    - Miss penalty – time to replace a block in the upper level + time to deliver the block to processor
- Average access time = **Hit time + Miss penalty x Miss rate**
- Cache line
    - Block of data copied from RAM
    - Address tag for the starting memory location of cache entry
- Cache and CPU operate
    - CPU requests content of memory location
    - Cache check first for this data
    - If data in cache then **HIT**
    - If not in cache then **MISS** issue READ for the block from Main memory to cache, then copy the cache to CPU
    - Cache includes **tags** to show which block of main memory is in each cache slot
- Direct-Mapped Cache
    - Each memory location mapped to exactly 1 cache location
    - Uses [block address in Main memory] MOD [Number of cache blocks]
    - Tag indicate block group: slides 32
- Fully associative Cache
    - A block is placed at **any** location in cache, no cache index needed, reduce miss ratio
    - Search all blocks in the cache
    - Search done in parallel
    - Only good for cache with small number of blocks
    - Tag indicate block address: slide 34
- Set-associative Cache
    - Combine direct and full associative
    - Mapping of a block from memory to a set and any set to any position in cache
- **T average (access time from CPU) = hit rate * cache access time + (1-hit rate) * miss penalty**
- Improve hit rate
    - Block sizes not too small and not too large
-

Instruction Pipeline

- Pipelining is an implementation technique where multiple instructions are overlapped in execution
- All Pipelining stages operate simultaneously
- Time for executing a single instruction is **not** shorter for pipelining

- Pipelining is faster for executing many instructions because everything works in parallel
- **All pipeline stages take a single clock cycle. Clock cycle must be long enough for the slowest operation**
- Pipelining – increasing performance
    - Sequential execution – Fetch cycle, Execute cycle
- Superscalar processor – adding multiple processing units for several instructions in parallel in each stage (not the same as dual core)
- Performance issues
    - Need to execute N processes
    - Each process takes m cycles
    - **Total serial = N * m cycles**
    - **Total pipeline = m + N-1**
        - First item after m cycles
        - Second item after m+1 cycles
        - N-1 cycles for last item
- Speedup
    - **Speedup = Tserial / Tpipeline = (N * m) / (m+N-1)**
    - It will never exceed the value of m
- **Throughput: number of instructions executed per second**
    - **Ps = R / S**
- Pipelining increases performance by increasing instruction throughput
- Pipeline hazards
    - Structural hazards
        - Hardware can't support the combination of instructions to execute in same clock cycle
    - Data hazards
        - When instructions depend on the results of a previous instruction in the pipeline. Need to stall the pipeline
    - Control hazards
        - Need to make a decision based on the results of one instruction while others are executing (ex: branch instruction)
    - Instruction hazard – when one of the instructions is not yet available
- Branches on performance
    - Penalty b = c cycles
    - P_b = probability that instruction is a branch
    - P_t = probability that a branch is taken
    - **CPI average = 1 + b * P_b * P_t**
    - **Execution efficiency = CPI no branch / CPI average**
        - when there **are** branches
- Pipelining improves performance by increasing instruction throughput
- Pipelining is a technique that exploits parallelism among the instructions in a sequential instruction stream

Virtual Memory

- Memory management technique that virtualize all storage and makes them appear as one large memory
- Efficient and safe sharing of memory among multiple programs

- Decrease problem of small memory space
- Virtual memory gives an illusion of an unbounded amount of memory
- Provide relocation
- How?
    - CPU produce virtual address
    - Hardware/software support translation
    - Become a physical address
- Virtual memory resides in the hard disk
    - RAM acts as a cache and takes a chunk from hard disk
    - RAM for **active** code and data while everything else on hard disk
- Memory management Unit (MMU) – translates virtual address to physical address
- Swap space
    - Virtual memory stored in hard disk "swap space"
    - Physical memory holds a small number of virtual pages
    - Mapping between virtual and physical memory
- **Page Table is in RAM –** records the mapping of virtual pages to physical frames
- Max RAM we can have = 2^(page frame) * page size
- Max VM we can have = 2^(disk location) * page size
- Page Table entries (2^n), n = 2^(address space) / 2^(page size)
- Page faults – data is not in memory, get it from hard disk
    - Huge miss penalty
    - Must reduce fault rate
- Translation lookaside buffer (TLB)
    - Used to speed virtual address translation
    - Acts like a cache of page table entries
-