

TSTE12 Design of Digital Systems Lecture 12

Kent Palmkvist



Agenda

- Microprocessor structures and programming
- Assembly language
- C-language low level programming

Practical issues

- Project presentation no later than 28/10
- I have not checked exams, may require an earlier date if exams 28/10
- Two sessions
 - 2 groups/session
- 1 group presents while others are acting as audience, then swap
- 20 minutes for each group, including demo
- Projector, DE2-board, screen, keyboard, speakers available in presentation room.

Microprocessor usage

- Suitable for complex programming
 - User interfaces
 - Complex state machine behavior
- Standard components
- Longer response time
 - Responses in range of us, ms, or more
- High resource utilization
 - ALU, registers etc.
- Sequential processing

Why leave microprogrammed structures

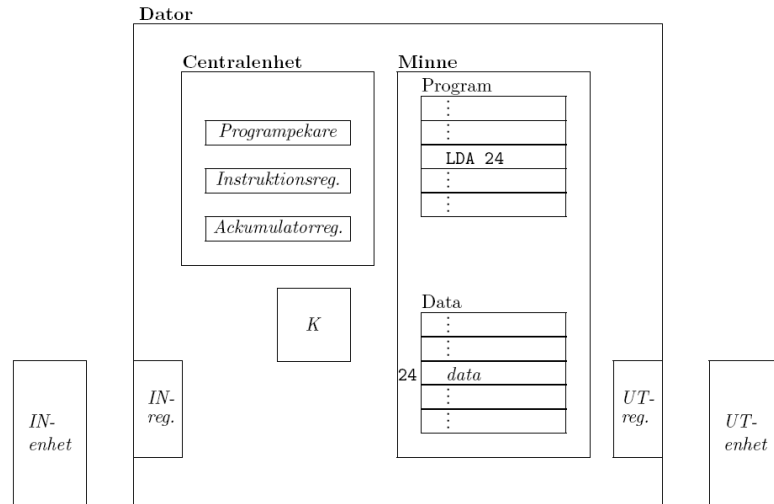
- Assembly language simplifies programming
 - No need to understand all small details
 - Lot of timing issues hidden
- Smaller memory footprint than microprogrammed
 - Previous microprogrammed example: long sequence of event for loading register value
 - Many control bits never used at the same time

Why leave microprogrammed structures, cont.

- Increase reuse
 - Architecture of processor may change while keeping the assembly language format
 - Example: 8086->80386->pentium->core2->i7
 - Sometimes binary compatible
 - Compilers of high-level languages
 - C/C++, JAVA, Python, Perl,....

Model computer example

- Computer
 - Central processing
 - Program pointer
 - Instruction regist
 - Ackumulator
 - Memory
 - Program
 - Data
 - Peripherals
 - Inputs
 - outputs



General Microprocessor Structure

- Similar to microprogrammed structure
- Program information stored in memory
 - Shared with data contents
- Program counter
 - Point to next instruction to execute
- Instruction Register
 - Current executed instruction (not visible to programmer)

Programmer model

- General Purpose Registers
 - Single or multiple registers
- Special purpose registers
 - Program counter (PC)
 - Stack pointer (SP)
 - Index registers (addressing modes)
 - Flag register
- Memory space
 - Read or write to memory cells
 - Some addresses does not have memory cells

Microprocessor behavior

- Fetch
 - Read Program instruction from memory (pointed to by program counter (PC) register)
- Decode
 - Determine what to happen, create control signals,
- Execute
 - Update register values, move data to/from memory, arithmetic/logic operations, jumps, M

Assembly level programming

- Describe each instruction used to implement behavior
 - Work on internal registers and/or memory cells
- Platform dependent
 - Each processor family have their own instruction set
 - Many models of the same CPU family share instruction set (e.g., 8086 - core i7)
- Maximum detail (compared to C etc.)

Instruction types

- Memory access
 - Includes I/O input and output
 - Support various addressing modes
- Arithmetic and logic
 - Modify/calculate register values
 - Include shift and rotate
- Register transfer
 - Move values between registers

Instruction types, cont.

- Branch and Jump
 - Includes conditional branch/jump
- Stack, Subroutines
 - pop/push, call, return from subroutine
- Control
 - Enable/disable interrupts, hardware breakpoints etc

Adressing modes

- Immediate
 - Data in instruction itself
- Direct
 - Address defined in instruction
- Indirect
 - Register contains address to use
- Indexed
 - Address plus offset

Program flow

- Very similar to microcode
 - Single sequential execution of instructions
 - Branch/jump used to implement loops, conditional statements
- Subroutines implements function calls
 - Subroutine call saves next instructions location before jump to subroutine
 - At end of subroutine restore PC to make jump back to instruction after subroutine call

Interrupts

- Give response without polling/checking continuously
- Interrupt sequence due to external event
 - Timer, I/O, Illegal instruction, etc.
- Interrupt routine at predefined location in memory
- Sequence being interrupted must not notice interrupt
 - Save processor state, and restore after completed interrupt routine
 - Similar to a subroutine call, but without any instruction making the call

C-level programming

- Platform independent or with little platform dependence
 - Big endian vs little endian
 - Word size (8, 16, 32, 64)
- Possible to describe interrupt routines etc (same as assembly language)
- Use of hardware through memory mapped I/O
 - Store values into registers
 - Read values from registers

C-level programming, cont.

- Registers in the processor not directly accessible
 - Compiler decides where to put variables (registers, memory etc.)
- Simple constructs may be translated into long sequences of assembly code
- Less control of code
- Possible to mix with assembly language

I/O example

- Parallel input port for switches
 - Decode memory address, read value directly
- Parallel output port for LED

- Write to a register driving the LEDs

```
#define SWITCHES_BASE_ADDRESS 0x10000010
#define LEDR_BASE_ADDRESS 0x10001000
```

- Pointers used to reference memory

```
int main(void)
{
    int * red_leds = (int *) LEDR_BASE_ADDRESS;          /* red_leds is a pointer to the LEDRs */
    volatile int * switches = (int *) SWITCHES_BASE_ADDRESS; /* switches point to toggle switches */
    while(1)
    {
        *(red_leds) = *(switches);                      /* Red LEDR[k] is set equal to SW[k] */
    }
    return 0;
}
```

Additional subjects

- Floating point calculations and hardware
- Caches
- Virtual memory
-

