Introduction to Assembler

Microcontroller VL
Thomas Nowak
TU Wien
Overview

- Introduction
- Simple Processor Architecture
- Addressing Modes
- Instructions
- Examples
- Guidelines
- Summary
Introduction

• Why should I learn assembler?
  – to see how all programs you write “really” end up
  – to understand the CPU architecture better
  – to see where speed improvements are possible
  – to realize there is no big secret behind it all
  – to pass the course ;)}
Introduction

• Assembler is basically a 1-1 mapping to machine code (with a few twists)
• Assembly language is human readable
  – machine code is not
• No high-level language constructs, e.g. if, while, ...
  – how to reassemble these constructs: later in this presentation (Examples)
• No nested expressions
  – e.g. you cannot write \texttt{add (mult 3,2), 1}
Introduction

• Operate directly on processor registers
  – no local variables
• Memory access is not transparent
  – different instructions for different locations
• No standardized instruction set
  – different instruction sets on different computer architectures
  – highly hardware dependent
Simple Processor Architecture

- **n General Purpose Registers (GPRs)**
- **Program Counter (PC)**
  - stores address of current instruction
- **Stack Pointer (SP)**
  - points to top of stack (TOS)
  - both possible: TOS or TOS +1 stored in SP
- **Status Word (SW)**
  - stores status of last instruction
Simple Processor Architecture

Program Code

0x00
..  
0x41: call 0x93  
0x42
..  
0x93
..  
0xFF

Main Memory

0x00: 0x10
0x01: 0x04
..  
0xFD: 0x74
0xFE: 0x42
0xFF: 0x00

General Purpose Registers (GPRs)

R1 ... Rn

Program Counter (PC)

Stack Pointer (SP)

Status Word (SW)
Addressing Modes

- code addressing (for jump instructions)
  - absolute
  - relative
  - indirect

- data addressing (for memory access)
  - direct
  - indirect (register/memory)
  - autoincrement/-decrement
  - etc.
Addressing Modes

• code addressing
  – you can define labels to code addresses
  – “jmp label3” instead of “jmp 0x04F3”

• direct addressing
  – effective address is the given address
  – “ld r1, 0x4FF3” loads the content of 0x4FF3 into register r1

• indirect addressing
  – effective address is the content of the given register/memory location
Addressing Modes

- autoincrement
  - indirect addressing mode that alters the indirecting register/memory location
  - “ld r1, r5+” loads contents of r5's value into r1 and increments r5

- addresses are often written as offset(base), i.e. address = base + offset
  - “ld r1, -8(r5)” loads contents of memory location ((r5's value) – 8) into register r1
Instructions

• Three styles of binary operators
  – OP src1, src2, dst
    i.e. dst := src1 OP src2
  – OP r1, r2
    i.e. r1 := r1 OP r2
  – OP r1 (other operand in AC register)

• Two styles of comparing registers
  – CMP r1, r2, dst
    i.e. dst := r1 CMP r2
  – having the status word updated after certain operations (e.g. negative and zero bit)
Instructions

• Arithmetic/logical instructions
  – ADD, SUB, AND, NOT, ...
  – different instructions for integer and float registers

• Jump instructions
  – conditional (e.g. BREQ) and unconditional (BR) branches
  – jump to subroutine (JSR) saves the current program count (PC) to a register or stack (RA)
  – RET jumps to address in RA
Instructions

• Load/store instructions
  – LD, ST
  – many addressing modes possible (CISC CPUs)
  – transparent stack instructions possible (PUSH, POP)
  – LDI r1, value (Load Immediate) assigns value to register r1

• Move instruction (MOV)
  – MOV r1, r2 copies r2's value to r1
Instructions

• Compare instructions
  – combined compare/branch instructions possible
    Example: branch if r1's value is greater than r2's (BRGT r1, r2, label)
Examples

• Implementing an if-statement in Assembler 1/2

```c
if(r1==2)
    r2=0;
else
    r2=1;
```

• r1 and r2 are CPU registers here
Examples

• Implementing an if-statement in Assembler 2/2

\[
\begin{align*}
\text{ldi} & \ r3, \ 2 && r3 := 2; \\
\text{cmp} & \ r1, \ r3 && r1 - r3; \quad \text{//update SW} \\
\text{brne} & \text{ else} && \text{if(!zero\_flag) goto else;} \\
\text{ldi} & \ r2, \ 0 && r2 := 0; \\
\text{jmp} & \text{ end} && \text{goto end;} \\
\text{else:} & \text{ else:} \\
\text{ldi} & \ r2, \ 1 && r2 := 1; \\
\text{end:} & \text{ end:}
\end{align*}
\]
Examples

• Implementing a while-loop in Assembler

```c
while (r1 < 20)
    ++r1;
```

• r1 is a CPU register
Examples

• Implementing a while-loop in Assembler 2/2

while:
cmpi r1, 20
brge end
addi r1, 1
jmp while
end:

while:
r1 – 20;  //update SW
if(!negative_flag) goto
   end;
r1++;  
goto while;
end:
Examples

• Implementing a do-while-loop in Assembler 1/2

    do
        --r1;
    while (r1 > 5);

• r1 is a CPU register
Examples

• Implementing a do-while-loop in Assembler 2/2

```assembly
ldi r2, 5
do:
addi r1, (-1)
cmp r2, r1
brlt do
```

```
r2 := 5;
do:
r1--;
r2 – r1;    //update SW if(negative_flag) goto do;
```
Guidelines

• Look how the compiler transforms a high-level language program to assembler code

  - gcc -S code.c creates assembler code file code.s
  - this is a good source of negative examples
  - gcc allocates every variable on the stack, this is often not necessary
Guidelines

• Name used registers
  – use the .equ directive to create aliases for register numbers
  – if you use a register for different purposes in different parts of your program, give it multiple names
Guidelines

• Debug systematically
  – it is nearly impossible to code assembler “blindly”, i.e. without continuous testing
  – debug only small code blocks simultaneously
  – use LEDs to display current state while debugging
Guidelines

• Be redundant (sometimes)
  – assembler code is susceptible to hard-to-see mistakes
  – e.g. create redundant labels just to clarify the control flow:
    cpi r1, 1
    breq equals_one
    not_equals_one:
    ...
    equals_one:
    ...
  – label not_equals_one is redundant here, but documents the control flow
Summary

• Assembler is a 1-1 mapping to machine code
• Machine dependent instruction set
• Addressing
  – code
  – data
  – direct, indirect modes
Summary

• Groups of assembler commands
  – arithmetic/logical
  – jumps
  – load, store
  – compare
  – move