MeggyJava to AVR Assembly

Today

– MeggyJava class example
– MeggyJrSimple C++ library
– ATmega328p chip
– AVR assembly especially for PA3ifdots.java

Next week

– Post possible peer evaluation criteria for PA1 on the google form posted on piazza. (Potential extra credit).
– PA1: Start PA3whiledots.java.s. It is due in 13 days along with demo!
– HW2: Due Tuesday September 6th
  – Reading assignment before Thursday is posted on class schedule
  – Thursday will be Haskell intro
  – Friday will be Haskell in discussion section
Class Example

Let’s write a MeggyJava program
  – Infinite loop
  – If we press the A button then let’s light a pixel.
  – If we press the B button then …
  – How can we have variables?
  – …
Meggy Jr Simple Library

Key concepts

- LED screen (pixels)
- Auxiliary LEDs
- Buttons
- Speaker

- Check the AVR-G++ generated code for library calls, and their calling sequence. AVR-G++ (and also MeggyJava) links in run time libraries:
- **Meggy Jr** Library provided an interface to set and read values in the Display Memory
- **Meggy Jr Simple** lies on top of Meggy Jr library, and provides a higher level API with names for e.g. colors
- Michelle Strout and students (honors projects / theses) added some functionality to the Meggy Jr Simple library
Meggy Jr Simple Library functions

ClearSlate() -- erase the whole slate
DrawPx(x,y,color) -- set pixel (x,y) to color
DisplaySlate() -- copy slate to LED Display Memory
SetAuxLEDS(value)
    -- 8 LEDs above screen numbered 1, 2, 4,..,128 (left to right)
        value is a byte encoding in binary which LEDs are set
        SETAuxLEDS(53) sets LEDs 1, 4, 16, and 32
ReadPx(x,y) -- returns byte value of pixel (x,y)

CheckButtonsDown()
    -- sets 6 variables: Button_(A|B|Up|Down|Left|Right)
GetButtons() returns a byte (B,A,Up,Down,Left,Right: 1,2,4,8,16,32)
ToneStart(divisor, duration)
    -- starts a tone of frequency 8 Mhz/divisor for ~duration milliseconds
        There are predefined tones.
Check out MeggyJrSimple.h
Example AVR-G++ program

/* 1/24/11, MS, goal is to exercise all of the routines in MeggyJrSimple */
#include "MeggyJrSimple.h"
#include <util/delay.h>

int main (void) {
    MeggyJrSimpleSetup();
    DrawPx(0, 1, Red);  // should display red LED
    DisplaySlate();
    // If <0,1> pixel is red, set auxiliary light
    if (ReadPx(0,1)==Red) { SetAuxLEDs (4); }
    while (1){
        CheckButtonsDown();
        if (Button_A) {  Tone_Start(ToneC3, 1000); }
        if (Button_B) { SetAuxLEDs(16); }
        if (4 & GetButtons()) { SetAuxLEDs(31); }  //
        if (Button_Up) { delay_ms(256); }
    }
    return 0;
}
AVR Instruction Set Architecture, or Assembly

ATmega328p

AVR ISA

Handling GetButton and SetPixel calls, (Calling Convention)

Handling if statements (Condition Codes and Branches)

Handling expression evaluation (Operations and Stack instructions)

Variables on the stack and in the heap
ATmega328p

Terminology
- Atmel, a company
- AVR, 8-bit RISC instruction set architecture for a microcontroller
- ATmega328p, AT for Atmel, MegaAVR microcontroller, 32kb flash, 8-bit AVR, p=low power
- Arduino, programming environment for various boards with some AVR chips

Uses
- Very popular for hobbyists
  http://www.engineersgarage.com/articles/avr-microcontroller
- Industry: Whirlpool appliances, electric car charger, medical products, …
AVR Instruction Set Architecture (ISA)

AVR is an 8-bit (1 byte) Harvard RISC Architecture
  – Two 8-bit words (and register pairs e.g. R0, R1) can be interpreted as 16-bit ints

Harvard: There are separate spaces in memory
  – data space (data) (0-RAMEND)
  – program space (text) (0-FLASHEND)

There are 32 Registers, organized in a register file R0 – R31

There is a run time Stack (stack pointer/ push / pop) in dataspace

RISC: Reduced Instruction Set, What does it mean?
  – Only load/store instructions can access the memory
  – Most instructions work on registers only and have therefore fully predictable timing (#clocks to execute)
  – No self-modifying code.
Execution Model

PC →

text

ldi ...
add ...
sub ...

data

heap

↓

stack

↓

Registers

r0

r1

r2

r3

r31

stack pointer

← r29:r28

ALU
/**
 * PA3ifdots.java
 *
 * An example similar to the one for PA1.
 * The language features will be from the PA3 grammar.
 */

import meggy.Meggy;

class PA3ifdots {

    public static void main(String[] whatever){
        if (Meggy.checkButton(Meggy.Button.Up)) {
            Meggy.setPixel( (byte)3, (byte)(4+3), Meggy.Color.BLUE );
        }
        if (Meggy.checkButton(Meggy.Button.Down)) {
            Meggy.setPixel( (byte)3, (byte)0, Meggy.Color.RED );
        }
    }
}
Calling convention

Calling convention is interface between caller and callee
- callers have to pass parameters to callee
- callees have to pass return values to caller
- callers and callees save registers
  caller saves registers r18-r27, r30-r31
  callee saves registers r2-r17, r28-r29
- Arguments - allocated left to right, r25 to r8
  r24, r25 parameter 1, only use r24 if just a byte parameter
  r22, r23 parameter 2
  … r8, r9 parameter 9

Return values
  8-bit in r24, 16-bit in r25:r24,
  up to 32 bits in r22-r25, up to 64 bits in r18-r25.
Meggy Java program for translation to AVR (calls)

/* PA2bluedot.java */
import meggy.Meggy;

class PA2bluedot {
    public static void main(String[] whatever){
        Meggy.setPixel( (byte)1, (byte)2, Meggy.Color.BLUE );
    }
}

/* prologue: function */
.file "main.java" /* frame size = 0 */
__SREG__ = 0x3f
__SP_H__ = 0x3e
__SP_L__ = 0x3d
__CCP__  = 0x34
__tmp_reg__ = 0
__zero_reg__ = 1

.global __do_copy_data
.global __do_clear_bss
.text
.global main
.type main, @function

main:
    call _Z18MeggyJrSimpleSetupv
    ldi r24,lo8(1)
    ldi r22,lo8(2)
    ldi r20,lo8(5)
    call _Z6DrawPxhhh
    call _Z12DisplaySlatev
    jmp endLabel
    ret
.endLabel:
    .size main, .-main

AVR Assembly
/**
 * PA3ifdots.java
 *
 * An example for the students to code up in AVR assembly for PA1.
 * The language features will be from the PA3 grammar.
 */

import meggy.Meggy;

class PA3ifdots {

    public static void main(String[] whatever) {
        if (Meggy.checkButton(Meggy.Button.Up)) {
            Meggy.setPixel((byte)3, (byte)(4+3), Meggy.Color.BLUE);
        }
        if (Meggy.checkButton(Meggy.Button.Down)) {
            Meggy.setPixel((byte)3, (byte)0, Meggy.Color.RED);
        }
    }
}

Meggy Java program for translation to AVR (if statement)
AVR Status Register

**Status Register (SREG)** keeps some bits (flags) that represent an effect of a previously executed instruction

Some important flags (there are more, check the Atmel AVR manual)

- **C**: Carry flag, a carry occurred (bit overflow)
- **Z**: Zero flag, result was 0
- **N**: Negative flag, result was negative
  - The effect on flags by instruction execution can be cleared (0), set (1), unaffected (-)

Conditional Branch instructions (breq, brlo, brlt, brne) use these flags

`brne label`
Flags and Conditional Branches

The comparison and arithmetic instructions set the flags $(Z,N,C,...)$

Comparison instructions: cp cpc tst

Arithmetic instructions:
adc add sbc sub neg and or eor lsl lsr muls rol ror

Conditional branch instructions inspect the flags:

Branch instructions: brlo brlt brmi brne

Branches branch PC relative and have a limited range (-64 .. 63)
Therefore, if we don’t know how far a branch will branch, we need to branch to a jump instruction (jmp), which can reach all instructions
/* PA5movedot.java */
...
if (Meggy.checkButton(Meggy.Button.Up)) {
    this.movedot(curr_x, (byte)(curr_y+(byte)1));
    Meggy.toneStart(localvar, 50);
} else {}
/**
 * PA3ifdots.java
 * 
 * An example for the students to code up in AVR assembly for PA1.
 * The language features will be from the PA3 grammar.
 */

import meggy.Meggy;

class PA3ifdots {

    public static void main(String[] whatever){
        if (Meggy.checkButton(Meggy.Button.Up)) {
            Meggy.setPixel( (byte)3, (byte)(4+3), Meggy.Color.BLUE );
        }
        if (Meggy.checkButton(Meggy.Button.Down)) {
            Meggy.setPixel( (byte)3, (byte)0, Meggy.Color.RED );
        }
    }
}
Arithmetic: bytes and ints

AVR is an 8 bit architecture, but has support for 16 bit ints.

This is accomplished by having register pairs, and having certain instructions taking certain flags into account:

# add r1:r0 to r3:r2
    add r2,r0  # Rd = Rd + Rr  sets C
    adc  r3,r1  # Rd = Rd + Rr + C

Subtraction: check out “sub” and “sbc”
Multiplication: check out “muls”
Bitwise AND: check out “and”
Meggy Java program for translation to AVR (expression eval)

/* PA5movedot.java */
...
return ((byte)(0-1) < x) ...

.file "PA5movedot.java"

# Load constant int 0
ldi r24,lo8(0)
ldi r25,hi8(0)
# push two byte expression onto stack
push r25
push r24

# Load constant int 1
ldi r24,lo8(1)
ldi r25,hi8(1)
# push two byte expression onto stack
push r25
push r24

# load a two byte expression off stack
pop r24
pop r25

# Do INT sub operation
sub r24, r18
sbc r25, r19
# push hi order byte first
# push two byte expression onto stack
push r25
push r24

# Casting int to byte by popping
# 2 bytes off stack and only push low bits
# back on. Low bits are on top of stack.
pop r24
pop r25
push r24
Variables on the Stack and Heap

- **text**
  - `ldi ...
  - `add ...
  - `sub ...

- **data**
  - `heap
  - `stack

- ** Registers**
  - `r0
  - `r1
  - `r2
  - `r3
  - `r31

- **PC →**
- **stack pointer ← r29:r28**

- **ALU**
Stack and heap

**Stack pointer:**
points at first available location on the run time stack
varies during expression evaluation

**Frame pointer:**
a fixed pointer in the stack frame so that parameters and local
variables can be associated with an offset from the frame pointer

Allocating space on the heap with **malloc library function:**
malloc allocates $n$ consecutive bytes in the heap and
returns the address of the first byte allocated.
(Will see examples of this later).
Data Indirect addressing

Some register pairs are used for indirect addressing.
There are special names for these Indirect Address Registers

\[ X=R27:R26, \ Y=R29:R28, \ Z=R31:R30 \]

```
in r28,___SP_L___  // putting the stack pointer into r29:r28
in r29,___SP_H___
```

```
ldd \ r24, Y+3 \ // load byte that is 3 bytes from address in r29:r28
// r24 = M[r29:r28 + 3]
```

```
std \ Y+1, r24 \ // store value in r24 to address r29:r28+1
// M[r29:r28 + 1] = r24
```

There are pre-decrement and post-increment indirect addressing modes for data structure (Stack) manipulation

The run time stack is implicitly manipulated with (push) and (pop) instructions, SP is the name of the stack pointer
Before Next Time

Post possible peer evaluation criteria for PA1 on the google form posted on piazza. (Potential extra credit).

PA1: Start PA3whiledots.java.s. It is due in 13 days along with demo!

HW2: Due Tuesday September 6th
   – Reading assignment before Thursday is posted on class schedule
   – Thursday will be Haskell intro
   – Friday will be Haskell in discussion section