Meggy Jr Simple and AVR

Today

- Meggy Jr Simple library
- ATmega328p chip
- avr assembly especially for PA3ifdots.java

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;
 }
```

Mapping Meggy Java Interface to Meggy Simple Interface

Let's look at some examples of how this works.

AVR Instruction Set Architecture, or Assembly

ATmega328p

Why assembly?

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://hacknmod.com/hack/top-40-arduino-projects-of-the-web/

http://www.engineersgarage.com/articles/avr-microcontroller

- Industry: Whirlpool appliances, electric car charger, medical products, …

Why Assembly?

It is the target language for (C++, MeggyJava) compilers, so they can generate symbolic code, and don't need to

resolve (references to) labels, linking

create .hex files

We can link the C++ run time Meggy Jr libraries

Assembly programming:

For some embedded processors, still need to do some assembly programming (e.g. device drivers).

We want to understand / express how the run-time stack works

AVR Instruction Set Architecture (ISA)

AVR is an 8-bit (byte) Harvard RISC Architecture

Two 8-bit words (and register pairs e.g. R0, R1) can be interpreted as 16 bits ints

Harvard: There are separate spaces

- 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)

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)

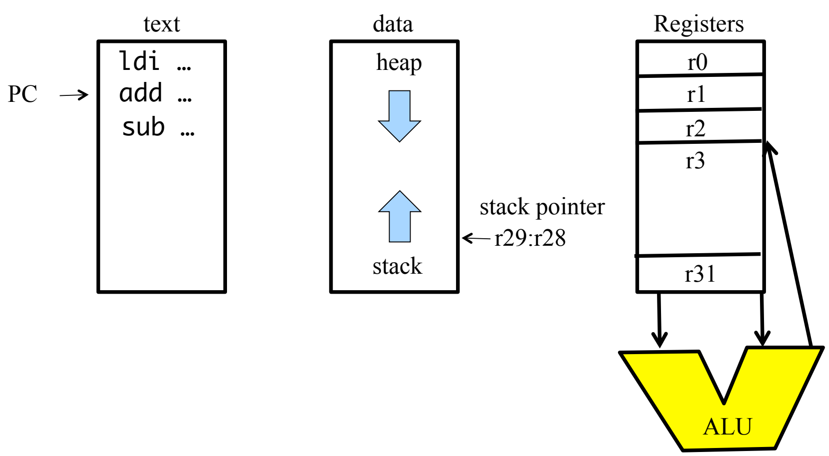
Addressing modes

Program and data addressing modes support access to the Program (flash) and Data memory (SRAM, Register file, I/O memory). See the AVR instruction Set document for the details

Instructions are packed in one or two words (2 bytes).

- **Direct register** uses the (names of) registers as operands
- **Data direct** has a 16-bit data address in the word following an instruction word
- **Relative** (PC relative) adds an offset to the program counter the offset has a limited range (-63 .. +64, or -2048..2047)

Execution Model



Meggy Java program for translation to AVR (calls)

```
/**
 * 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 );
        }
    }
}
```

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 );
        }
}
```

```
     .file    "PA2bluedot.cpp"
__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:
/* prologue: function */
/* frame size = 0 */
        call _Z18MeggyJrSimpleSetupv
        ldi r24,lo8(1)
        ldi r22,lo8(2)
        ldi r20,lo8(5)
        call _Z6DrawPxhhh
        call _Z12DisplaySlatev
.L2:
        jmp .L2
        .size   main, .-main
```


Meggy Java program for translation to AVR (if statement)

```
/**
 * 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 );
        }
    }
}
```


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

Meggy Java program for translation to AVR (if statement)

```
/* PA5movedot.java */
…
 if (Meggy.checkButton(Meggy.Button.Up)) {
   this.movedot(curr_x, (byte)(curr_y+(byte)1));
   Meggy.toneStart(localvar, 50);
 } else {}
```

```
…
 #### if statement

    ### MeggyCheckButton
    call    _Z16CheckButtonsDownv
    lds     r24, Button_Up
    # if is zero, push 0 else push 1
    tst     r24
    breq    MJ_L6
MJ_L7:
    ldi     r24, 1
    jmp     MJ_L8
MJ_L6:
MJ_L8:
```

```
    # push one byte expression onto stack
    push   r24
    # load condition and branch if false
    # load a one byte expression
    pop    r24
    #load zero into reg
    ldi    r25, 0
    #use cp to set SREG
    cp     r24, r25
    brne   MJ_L4
    jmp    MJ_L3

    # then label for if
MJ_L4:
    # then body …
    jmp    MJ_L5

    # else label for if
MJ_L3:
    # done label for if
MJ_L5:
```


Meggy Java program for translation to AVR (expression eval)

```
/**
 * 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

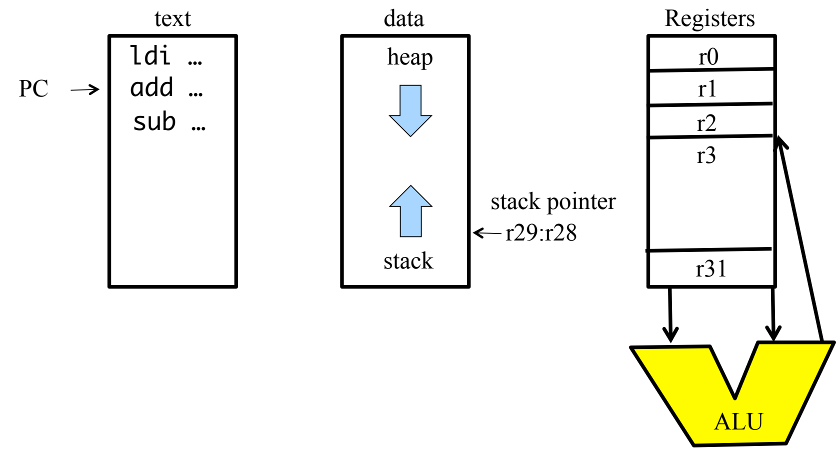
Meggy Java program for translation to AVR (expression eval)

```
/* 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    r18
 pop    r19
```

```
# 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



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 Registe**rs**

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