Assembly Language with GCC

Why use assembly language?

- -High level of control of code generation (don’t let the optimizer interpret)
- -Speed
- -Awkward C implementations (e.g., 16/24 bit SPI xfer for AD9851)

How is it done? Two ways...

- -Inline assembly
  - -assembly instructions written directly in the C code file
  - -downright weird syntax except for really simple stuff, e.g:

```
 asm volatile ("nop");  //inline assembly code, add a nop
asm volatile("sbi 0x18,0x07;");  //set some bits
```

- -Separate assembly code file
  - -.S file contains only assembly instructions
  - -just like writing assembly language programs but assembled and linked like another C file.

Assembly Language with GCC

Separate assembly code files

Discussing assembly language function calls here

Questions?....

-How do we pass in arguments to the function (where do they go?)
-How does the function pass back the return value (where is it?)
-What registers can be uses without saving?
-What registers must be saved?

Registers that can be used without saving first:

-r0, r1 (r1 must be cleared before returning)
-r18-r25,
-r26-r27 (X reg)
-r30-r31 (Z reg)

From the compiler's view, these *call used* registers can be allocated by gcc for local data. They may be used freely in subroutines.

Assembly Language with GCC

Separate assembly code files

Rules for r0 and r1

r1 maybe freely used within a function but it must be cleared before returning. “clr r1”.

ISRs save and clear r1 upon entering, and restore r1 upon exit in case it was non-zero at exit. r1 is assumed to be zero in any C code space.

r0 can be clobbered by any C code, its a temp register. It may be used "for a while" (from the documentation!) within the function call.

Assembly Language with GCC

Separate assembly code files

Registers that must be saved and restored by subroutines

r2 – r17
r28 - r29

Function call conventions – arguments

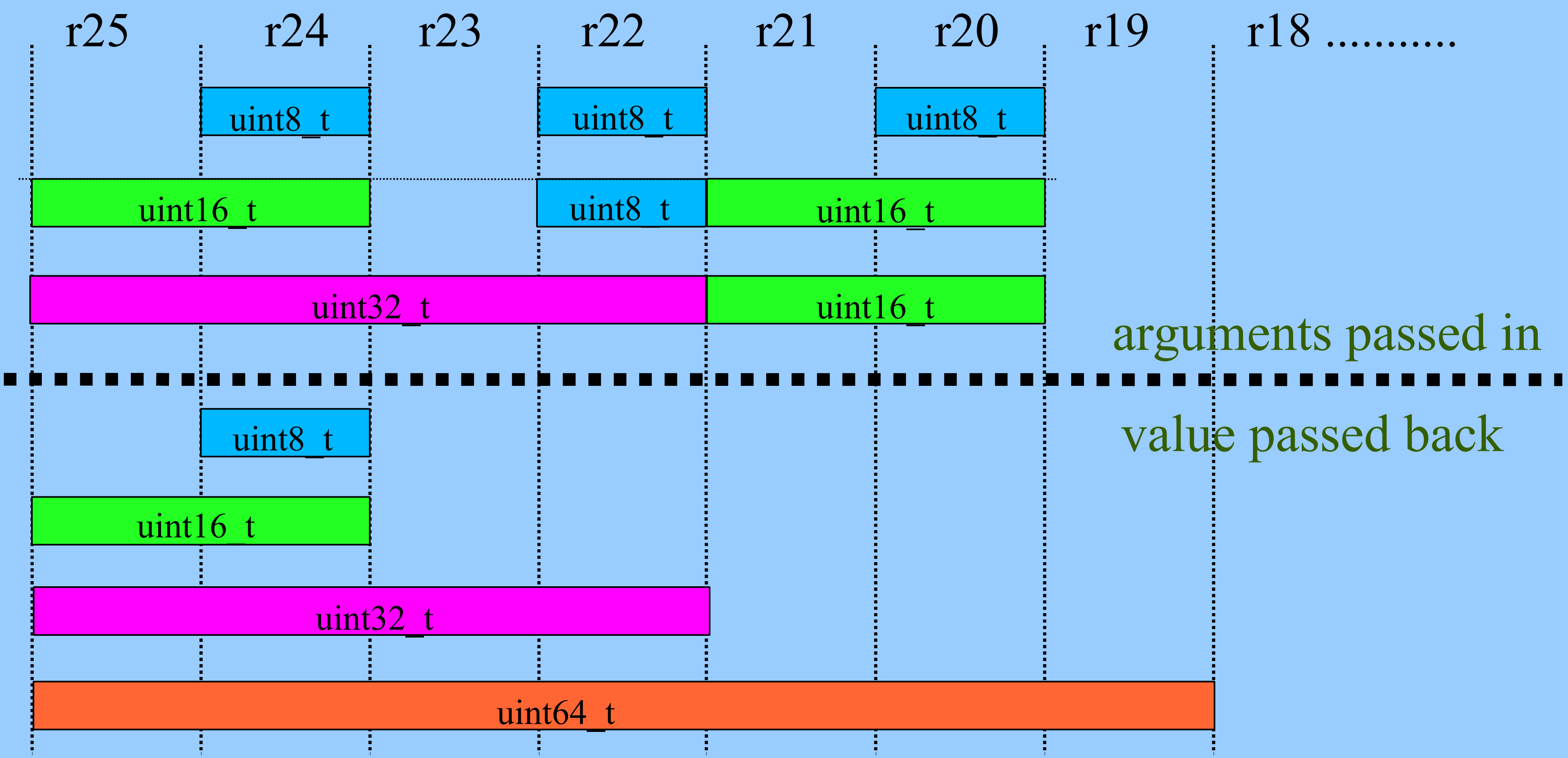
Arguments are allocated left to right, r25 to r18

All args are aligned to start in even numbered registers. Odd sized arguments like char, have one free register above them.

If there are too many arguments, those that don’t fit in registers are passed on the stack.

Assembly Language with GCC

Argument and value register convention for GCC



Assembly Language with GCC

Example: SPI function call in assembly language

```
//sw_spi.S, R. Traylor, 12.1.08

#include <avr/io.h>
.text
.global sw_spi

//define the pins and ports, using PB0,1,2
.equ  spi_port , 0x18  ;PORTB
.equ  mosi     ,    0  ;PB2 pin
.equ  sck      ,    1  ;PB0 pin
.equ  cs_n     ,    2  ;PB1 pin

//r18 counts to eight, r24 holds data byte passed in

sw_spi:   ldi r18,0x08         ;setup counter for eight clock pulses
          cbi spi_port, cs_n   ;set chip select low
loop:     rol r24              ;shift byte left (MSB first); carry set if bit7 is one
          brcc bit_low         ;if carry not true, bit was zero, not one
          sbi spi_port, mosi   ;set port data bit to one
          rjmp clock        ;ready for clock pulse
bit_low:  cbi spi_port, mosi   ;set port data bit to zero
clock:    sbi spi_port, sck    ;sck -> one
          cbi spi_port, sck    ;sck -> zero
          dec r18           ;decrement the bit counter
          brne loop         ;loop if not done
          sbi spi_port, cs_n   ;dessert chip select to high
          ret               ;
.end
```

Assembly Language with GCC

Example: SPI function call in assembly language

```
// assy_spi.c

#define F_CPU 16000000UL     //16Mhz clock
#include <avr/io.h>
#include <util/delay.h>

//declare the assembly language spi function routine
extern void sw_spi(uint8_t data);

int main(void)
{
  DDRB = 0x07;       //set port B bit 1,2,3 to all outputs
  while(1){
    sw_spi(0xA5);    //alternating pattern of lights to spi port
    sw_spi(0x5A);
  } //while
} //main
```

Assembly Language with GCC

Example: SPI function call in assembly language

```
PRG               = assy_spi
OBJ               = $(PRG).o  sw_spi.o
MCU_TARGET        = atmega128
OPTIMIZE          = -Os    # options are 1, 2, 3, s

DEFS              =
LIBS              =

CC                = avr-gcc

#override         CFLAGS        = -g -Wall $(OPTIMIZE) -mmcu=$(MCU_TARGET) $(DEFS)
override          LDFLAGS       = -Wl, -Map, $(PRG).map

OBJCOPY           = avr-objcopy
OBJDUMP           = avr-objdump

all: $(PRG).elf lst text eeprom

$(PRG).elf: $(OBJ)
$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $^ $(LIBS)

clean:
rm -rf *.o $(PRG).elf *.eps *.png *.pdf *.bak
rm -rf *.lst *.map $(EXTRA_CLEAN_FILES) *~

program: $(PRG).hex

sudo avrdude -p m128 -c usbasp -e -U flash:w:$(PRG).hex

lst:  $(PRG).lst

%.lst: %.elf
$(OBJDUMP) -h -S $< > $@
```