Assembly Language with GCC

- Why use assembly language?
  - High level of control of code generation
  - Speed...though questionable
  - Prevent awkward C implementations (16/24 bit xfers)
  - Possibly clearer implementations of some functions
- Here we discuss assembly language functions called by C code.
- This should be the typical case. You want to spend most of your time writing in a high-level language, not in assembler.
- Assembler code can call C code as well if necessary.

Assembly Language with GCC

- How is it done? - Two ways:
  - Assembly instructions can be written directly into C code.
  - A separate assembly file (.S) holds your assembly-only function.
- First, the simple case:

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

Assembly Language with GCC

- Now, for the other case, some questions arise...
- What registers can the function use without first saving?
  - R0, R1, but R1 must be cleared before returning
  - R18-R25
  - R26-R27 (X register)
  - R30-R31 (Z register)
- From the compiler's view, these "call used" registers can be used freely in the function call.
- Note: ISRs save and clear R1 upon entering, and restore R1 upon exit just in case it was non-zero at exit. R1 is assumed to be zero in any C code space.

Assembly Language with GCC

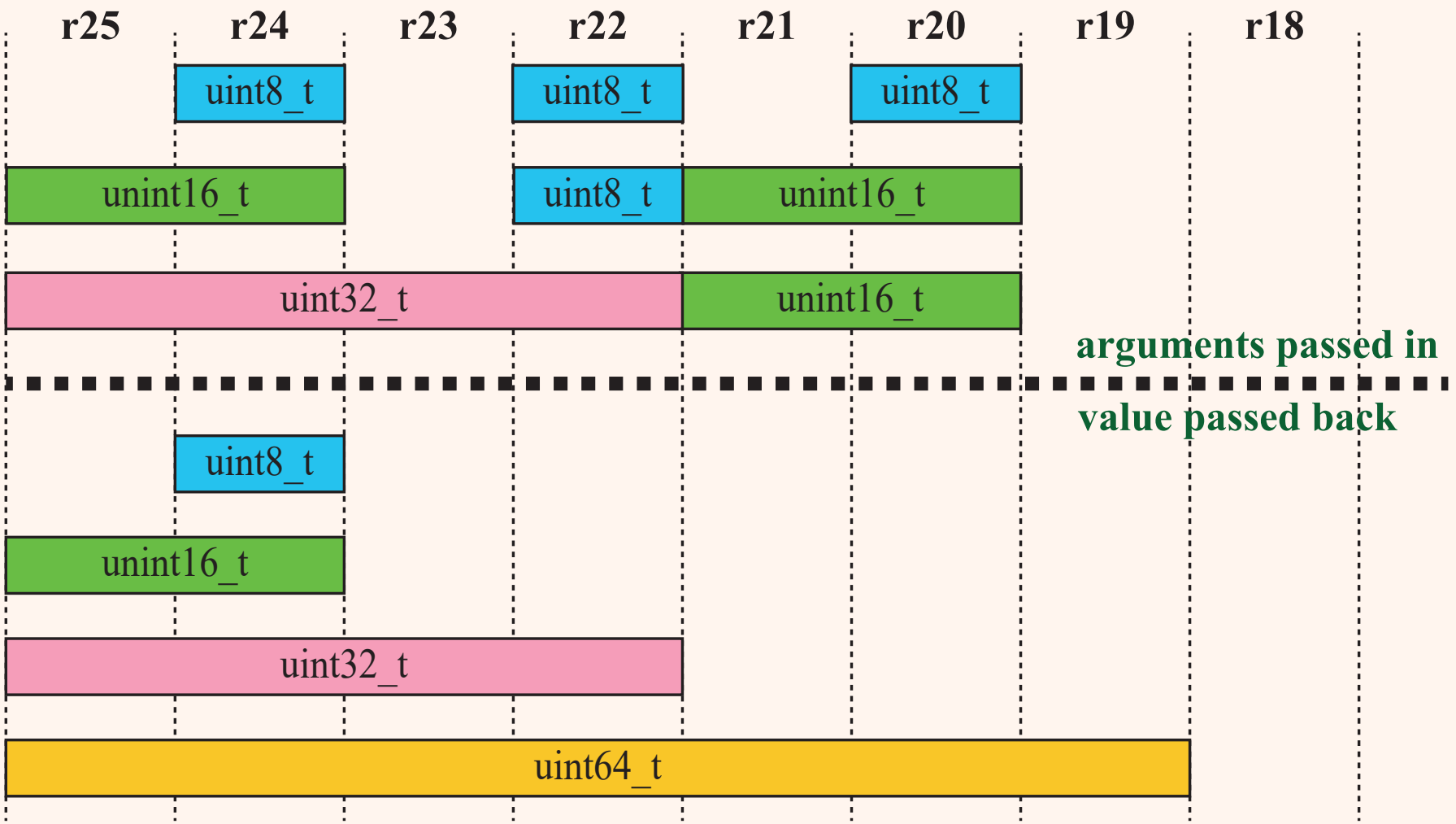
- ...some questions arise...
- What registers must be saved?
  - R2 through R17
  - R28 and R29

Assembly Language with GCC

- ...some questions arise...
  - How do we pass in arguments to and from the function?
    - 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 to many args, those that don't fit in registers are passed on the stack. Ouch!

Assembly Language with GCC

Passing arguments to and from the function



Assembly Language with GCC

```
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 cs_n,     2    ;PB1 pin

//r18 counts to eight, r24 holds data byte passed in
sw_spi:  ldi r18,0x08       ;setup counter for 8 clock pulses
cbi      spi_port,cs_n      ;set chip select low
loop:    rol r24            ;shift left; carry set if bit7=='1'
brcc     bit_low            ;if carry not true, bit7=='0',
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 '0'
clock:   sbi spi_port,sck   ;sck -> '1'
cbi      spi_port,sck       ;sck -> '0'
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

```
\\C code calling SPI function written in assembly

#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 sent to SPI port
    sw_spi (0x5A);
    _delay_ms(250);
  }//while
} // main
```

Assembly Language with GCC

```
PRG= assy_spi
OBJ= $(PRG).o sw_spi.o
MCU_TARGET = atmega128
OPTIMIZE   =-0s \# options are 1, 2, 3, s
DEFS =
LIBS =
#override CFLAGS  = -g -Wall \$(OPTIMIZE) -mmcu=$(MCU_TARGET) $(DEFS)
#override LDFLAGS = -w1, -Map, $(PRG).map
OBJCOPY = avr-objcopy
OBJDUMP = avr-objdump
all: $(PRG).elf 1st text eeprom
$(CC) $(CFLAGS) $(LDFLAGS) -o \$@ \$^ \ \$(LIBS)
clean:
rm -rf *.o $(PRG).elf *.eps *.png *.pdf *.bak
rn -rf *.1st *.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 \$< > \$@
```