Homework 2

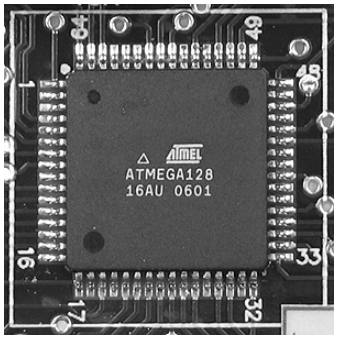
- Questions about it?

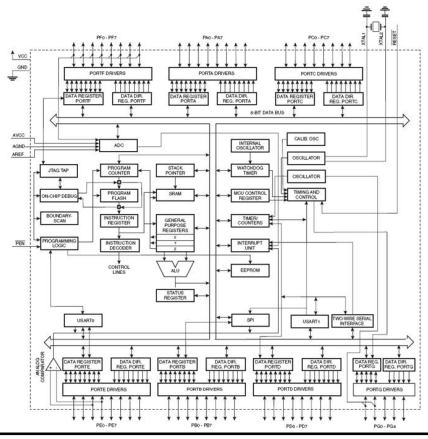
Last Time

- To write C and C++ programs that work, you have to understand a lot of subtle issues
  - Signed / unsigned rules
  - Sequence points
  - Implementation defined behavior
  - Unspecified behavior
  - Undefined behavior
- The C99 Standard is a free download
  - Google for n1124.pdf

Today

- Volatile
  - How to use it
  - How not to use it





- Example:
  - This AVR has no hardware multiply unit
  - Let's measure the speed of software multiply
- Solution using Timer 1:
  - Set timer to an appropriate rate
  - Read TCNT1
  - Do a multiply
  - Read TCNT1 again
  - Subtract first reading from second

Timer/Counter1 – TCNT1H and TCNT1L

Bit	7	6	5	4	3	2	1	0	
	TCNT1[15:8]								TCNT1H
	TCNT1[7:0]								TCNT1L
Read/Write	R/W	R/W	R/W	R/W	R/W	R/W	R/W	R/W	
Initial Value	0	0	0	0	0	0	0	0	

$31 ($51)	OCR0	Timer/Counter0 Output Compare Register							106
$30 ($50)	ASSR	–	–	–	–	AS0	TCN0UB	OCR0UB	TCR0UB	107
$2F ($4F)	TCCR1A	COM1A1	COM1A0	COM1B1	COM1B0	COM1C1	COM1C0	WGM11	WGM10	133
$2E ($4E)	TCCR1B	ICNC1	ICES1	–	–	WGM13	WGM12	CS12	CS11	CS10	136
$2D ($4D)	TCNT1H	Timer/Counter1 – Counter Register High Byte								138
$2C ($4C)	TCNT1L	Timer/Counter1 – Counter Register Low Byte								138
$2B ($4B)	OCR1AH	Timer/Counter1 – Output Compare Register A High Byte								138
$2A ($4A)	OCR1AL	Timer/Counter1 – Output Compare Register A Low Byte								138
$29 ($49)	OCR1BH	Timer/Counter1 – Output Compare Register B High Byte								138
$28 ($48)	OCR1BL	Timer/Counter1 – Output Compare Register B Low Byte								138
$27 ($47)	ICR1H	Timer/Counter1 – Input Capture Register High Byte								139
$26 ($46)	ICR1L	Timer/Counter1 – Input Capture Register Low Byte								139
$25 ($45)	TCCR2	FOC2	WGM20	COM21	COM20	WGM21	CS22	CS21	CS20	157
$24 ($44)	TCNT2	Timer/Counter2 (8 Bit)								159

```
#define TCNT1 (*(uint16_t *)(0x4C))

signed char a, b, c;

uint16_t time_mul (void)
{
  uint16_t first = TCNT1;
  c = a * b;
  uint16_t second = TCNT1;
  return second - first;
}
```

```
$ avr-gcc -Os -S -o - reg1.c
time_mul:
    lds r22,a
    lds r24,b
    rcall __mulqi3
    sts c,r24
    ldi r24,lo8(0)
    ldi r25,hi8(0)
    ret
```

```
#define TCNT1 (*(volatile uint16_t *)(0x4C))

signed char a, b, c;

uint16_t time_mul (void)
{
  uint16_t first = TCNT1;
  c = a * b;
  uint16_t second = TCNT1;
  return second - first;
}
```

```
avr-gcc -Os -S -o - reg2.c
time_mul:
    in r18,0x2c
    in r19,0x2d
    lds r22,a
    lds r24,b
    rcall __mulqi3
    sts c,r24
    in r24,0x2c
    in r25,0x2d
    sub r24,r18
    sbc r25,r19
    ret
```

In a header file…

```
// Timer/Counter 1
#define TCNT1 (*(volatile uint16_t *)(0x4C))

// T/C 1 Input Capture Register
#define ICR1  (*(volatile uint16_t *)(0x46))
#define ICR1L (*(volatile uint8_t *) (0x46))
#define ICR1H (*(volatile uint8_t *) (0x47))

// T/C 1 Output Compare Register
#define OCR1B (*(volatile uint16_t *)(0x48))
```

ColdFire Example

- C code you might write:

```
/* Enable signal as GPIO */
void make_pin0_gpio (void) {
  MCF_GPIO_PTCPAR = MCF_GPIO_PTCPAR_DTIN0_GPIO;
}
```

- Relevant reprocessor definitions:

```
typedef volatile uint8 vuint8;
#define MCF_GPIO_PTCPAR \
  (*(vuint8 *)(&__IPSBAR[0x10006F]))
#define MCF_GPIO_PTCPAR_DTIN0_GPIO (0)
```

ColdFire Example

- After the C processor has run, the code is:

```
void make_pin0_gpio (void) {
  (*(vuint8 *)(&__IPSBAR[0x10006F])) = (0);
}
```

- So, this is the code the CodeWarrior compiler actually sees and compiles
- Note: `vuint8 *` is pointer-to-volatile, not volatile-pointer
  - The distinction is crucial
- Hardware register access is typically done using pointers-to-volatile

Compiler Output

```
_make_pin0_gpio:
0x00000000  link     a6, #0
0x00000004  moveq    #0, d0
0x00000006  move.b   d0, ___IPSBAR+1048687
0x0000000C  unlk     a6
0x0000000E  rts
```

Another ColdFire Example

- C code you might write:

```
/* Enable signal as GPIO */
void make_pin0_gpio (void) {
  MCF_GPIO_PTCPAR |= MCF_GPIO_PTCPAR_DTIN0_GPIO;
}
```

- Expands out to:

```
void make_pin0_gpio_bogus (void) {
  (*(vuint8 *)(&__IPSBAR[0x10006F])) |= (0) ;
}
```

Another ColdFire Example

```
_make_pin0_gpio_bogus:
0x00000000  link     a6, #0
0x00000004  move.b   ___IPSBAR+1048687, d0
0x0000000A  unlk     a6
0x0000000C  rts
0x0000000E  nop
```

- What happened?
- Is the code what we wanted?
- Is the compiler correct?

Device Registers ≠ RAM

- Each read may return a different value
  - Free-running timer
- Writes may be ignored or result in undefined behavior
  - Read-only registers
- Reads can be writes
  - HCS12 interrupt flags cleared by writing 1
- Reads and writes can be side effecting
  - Launch a missile, raise the control rods, …

- RAM-like semantics are ingrained in language and compiler design
  - Useless loads eliminated
  - Redundant loads avoided by caching values in registers
  - Operations with constant arguments evaluated at compile time
  - Similar transformations for stores
- Memory behavior of optimized executable may be very different from source code

- Basic problem: Optimizations are in tension with HW register accesses
- Improving the optimizer breaks programs that previously worked
  - Lots of latent errors in real embedded programs
  - Problems not seen because compiler aren't smart enough
- Today we look at creating embedded systems that can't be broken by any future optimizer

- In early C there was no good solution

"At least one version of the UNIX Portable C Compiler (PCC) had a special hack to recognize constructs like

```
((struct xxx *)0177450)->zzz
```

as being potential references to I/O space (device registers) and would avoid excessive optimization involving such expressions"

- ANSI C (a.k.a. C89) added volatile
- Informal definition of volatile
  - Every read/write to a volatile variable that would be performed by a C interpreter must result in a load/store in the executable code, in the same order
  - Accesses shouldn't be added or removed
  - Accesses shouldn't be reordered (much)

- Volatile is a *type qualifier*
- Any type can be qualified
  - New types can be built from qualified types
- Rules for volatile are similar to, but not the same as, const

- Every level of indirection can be independently qualified:

```
int *p;
volatile int *p_to_vol;
int *volatile vol_p;
volatile int *volatile vol_p_to_vol;
```

- Making a struct or union volatile is same as making all members volatile
- Volatile bitfields are a little tricky

- Does this make sense?

```
const volatile int *p_to_const_vol;
```

- Yes: This is the correct declaration for a read-only timer register

Uses for Volatile

- Volatile use 1: HW register accesses
- Volatile use 2: Data shared between interrupts and main()
- Volatile use [illegible] ween threads
- Volatile [illegible]

- Now: Eight ways to create broken embedded code using volatile

#1: Not Enough Volatile

```
int done;

__attribute((signal)) void __vector_4 (void)
{
  done = 1;
}

void wait_for_done (void) {
  while (!done) ;
}
```

```
[regehr@babel ~]$ avr-gcc -Os wait.c -S -o -
__vector_4:
  push r0
  in r0,__SREG__
  push r0
  push r24
  ldi r24,lo8(1)
  sts done,r24
  pop r24
  pop r0
  out __SREG__,r0
  pop r0
  reti

wait_for_done:
  lds r24,done
.L3:
  tst r24
  breq .L3
  ret
```

Key property: Visibility

Make done volatile

```
wait_for_done:
.L3:
  lds r24,done
  tst r24
  breq .L3
  ret
```

#2: Too Much Volatile

- Some embedded developers make almost all globals volatile
  - Volatile is not a substitute for thinking
- Can seriously impact application performance
  - Hard to get the performance back since slow code is scattered everywhere

#3: Misplaced Qualifier

```
int *volatile REG = 0xfeed;
*REG = new_val;
```

- Oops!
- Typedefs are helpful:

```
typedef volatile int vint;
vint *REG = 0xfeed;
```

#4: Inconsistent Qualification

- In Linux 2.2.26
- arch/i386/kernel/smp.c:125

```
volatile unsigned long ipi_count;
```

- include/asm-i386/smp.h:178

```
extern unsigned long ipi_count;
```

- 2.3.x has similar problems
- Modern compilers will catch this

- Typecasts are another way to get inconsistent qualification
- Don't ignore compiler warnings about this!

#5: Ordering with Non-Volatile

```
volatile int ready;
int message[100];

void foo (int i)
{
  message[i/10] = 42;
  ready = 1;
}
```

```
$ gcc-4.3 -O3 barrier1.c -S -o -
foo:
        movl    4(%esp), %ecx
        movl    $1717986919, %edx
        movl    $1, ready
        movl    %ecx, %eax
        imull   %edx
        sarl    $31, %ecx
        sarl    $2, %edx
        subl    %ecx, %edx
        movl    $42, message(,%edx,4)
        ret
```

What happened?

- Non-volatile accesses can move around volatile accesses
- Volatile accesses cannot move around each other
  - Unless there are no intervening sequence points

- Solution 1: Make all shared variables volatile
- Solution 2: Use a "compiler barrier"

Compiler Barrier

- Tells the compiler:
  - No code motion past the barrier in either direction
  - Store all register values to RAM before the barrier
  - Reload values from RAM into registers after the barrier

GCC Barrier

```
volatile int ready;
int message[100];

void foo (int i)
{
  message[i/10] = 42;
  asm volatile ("" : : : "memory");
  ready = 1;
}
```

```
$ gcc-4.3 -O3 barrier2.c -S -o -
foo:
        movl    4(%esp), %ecx
        movl    $1717986919, %edx
        movl    %ecx, %eax
        imull   %edx
        sarl    $31, %ecx
        sarl    $2, %edx
        subl    %ecx, %edx
        movl    $42, message(,%edx,4)
        movl    $1, ready
        ret
```

- Compiler barriers are analogous to HW memory system barriers
- Not all compilers support barriers
  - CodeWarrior for ColdFire does not, unfortunately!
  - If not, inserting a call to an external function may work
- Good RTOS lock/unlock functions are compiler barriers
  - Often, only because they are not inlined
  - Problematic as compilers get smarter

Old Locks in TinyOS

```
char __nesc_atomic_start (void)
{
  char result = SREG;
  __nesc_disable_interrupt();
  return result;
}

void __nesc_atomic_end (char save)
{
  SREG = save;
}
```

New Locks in TinyOS

```
char__nesc_atomic_start(void)
{
  char result = SREG;
  __nesc_disable_interrupt();
  asm volatile("" : : : "memory");
  return result;
}

void __nesc_atomic_end(char save)
{
  asm volatile("" : : : "memory");
  SREG = save;
}
```

#6: Confuse Volatile and Atomic

- Accesses to volatile variables are not guaranteed to be atomic
- C doesn't guarantee atomicity of any access
  - However, char-, short-, and int-sized accesses are often atomic
- If you want atomicity, use a lock!

#7: Use Volatile on Modern Machines

- Volatile does not cause the compiler to emit memory fences or barriers
  - Consequently: Volatile is useless on out-of-order processors
- Volatile does not ensure visibility across a multiprocessor
  - Consequently: Volatile is useless on multicores

- Solution:
  - You must use a good lock implementation
  - These contain sufficient barriers make race-free programs execute in a sequentially consistent manner
- In a well-synchronized program volatile just slows it down and hides real problems
- Best not to hack your own synchronization primitives

#8: Assume the Compiler is Right

- Volatiles are frequently miscompiled

```
volatile int x;
void foo (void) {
  x = x;
}
$ msp430-gcc -O vol.c -S -o -
foo:
  ret
```

What does volatile really mean?

- We have to ask the standard
- Section 6.7.3 the C99 standard contains most of the details
  - "An object that has volatile-qualified type may be modified in ways unknown to the implementation or have other unknown side effects."

More Standard

- "What constitutes an access to an object that has volatile-qualified type is implementation-defined."
  - What??
- Apparently this is designed to cover the fact that some platforms have a minimum memory access width

Volatile Summary 1

- Volatile can be good
- You need it for:
  - Accessing device registers
  - Communicating with interrupts
- It's usually not useful for anything else
- Be careful about the compiler
  - CodeWarrior for ColdFire has volatile bugs
  - I've reported all of them that I'm aware of…

Volatile Summary 2

- Locks with compiler and HW barriers give atomicity and visibility
- Volatile does not
  - No atomicity
  - No visibility on advanced HW
- If you have good locks, use them instead of volatile for shared data