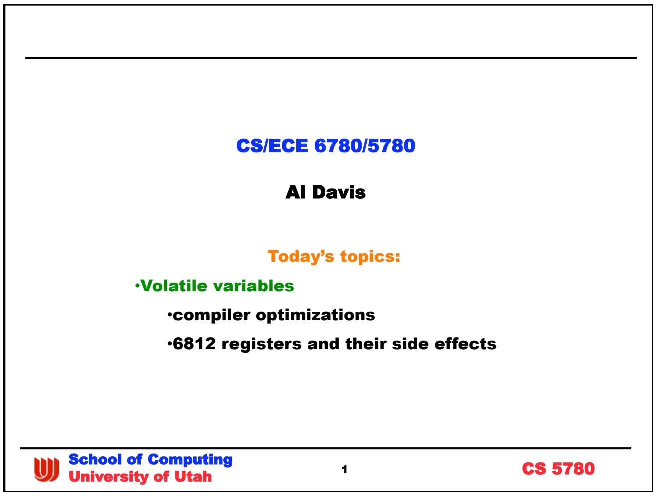
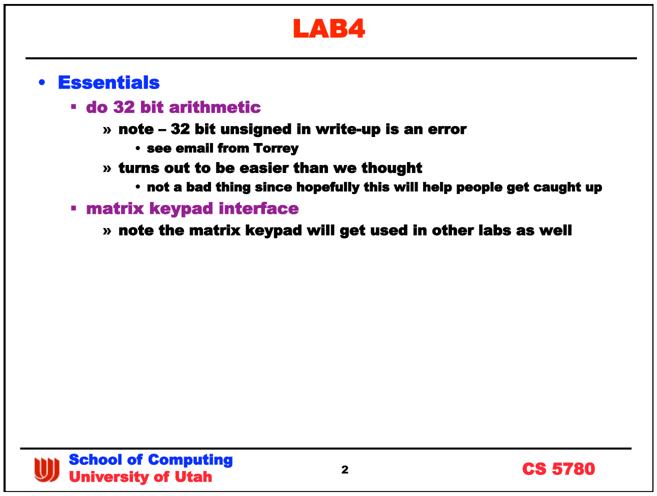


CS/ECE 6780/5780

Al Davis

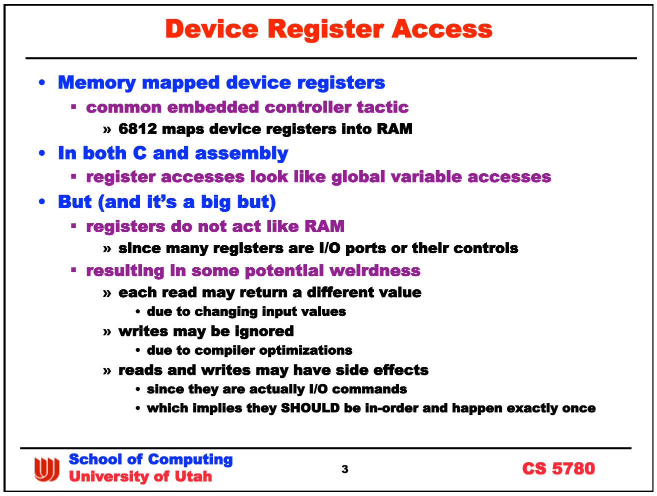
Today's topics:

- **Volatile variables**
  - **compiler optimizations**
  - **6812 registers and their side effects**



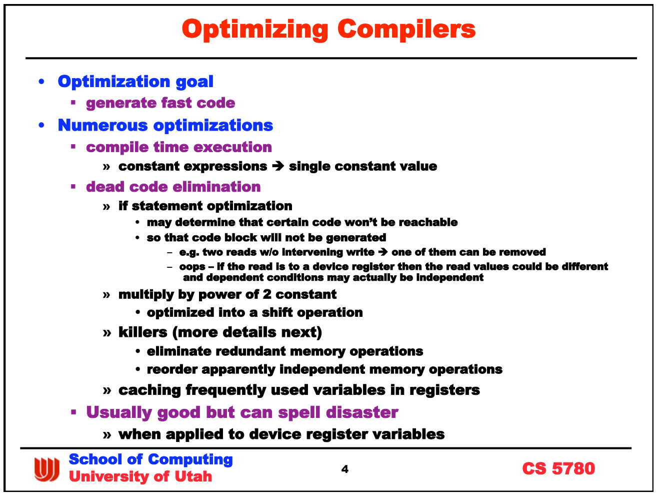
LAB4

- **Essentials**
  - **do 32 bit arithmetic**
    - » **note – 32 bit unsigned in write-up is an error**
      - **see email from Torrey**
    - » **turns out to be easier than we thought**
      - **not a bad thing since hopefully this will help people get caught up**
  - **matrix keypad interface**
    - » **note the matrix keypad will get used in other labs as well**



Device Register Access

- **Memory mapped device registers**
  - **common embedded controller tactic**
    - » **6812 maps device registers into RAM**
- **In both C and assembly**
  - **register accesses look like global variable accesses**
- **But (and it's a big but)**
  - **registers do not act like RAM**
    - » **since many registers are I/O ports or their controls**
  - **resulting in some potential weirdness**
    - » **each read may return a different value**
      - **due to changing input values**
    - » **writes may be ignored**
      - **due to compiler optimizations**
    - » **reads and writes may have side effects**
      - **since they are actually I/O commands**
      - **which implies they SHOULD be in-order and happen exactly once**



Optimizing Compilers

- **Optimization goal**
  - **generate fast code**
- **Numerous optimizations**
  - **compile time execution**
    - » **constant expressions → single constant value**
  - **dead code elimination**
    - » **if statement optimization**
      - **may determine that certain code won't be reachable**
      - **so that code block will not be generated**
        - – **e.g. two reads w/o intervening write → one of them can be removed**
        - – **oops – if the read is to a device register then the read values could be different and dependent conditions may actually be independent**
    - » **multiply by power of 2 constant**
      - **optimized into a shift operation**
    - » **killers (more details next)**
      - **eliminate redundant memory operations**
      - **reorder apparently independent memory operations**
    - » **caching frequently used variables in registers**
  - **Usually good but can spell disaster**
    - » **when applied to device register variables**

Memory Optimization Hazards

- **Eliminate redundant memory operations**
  - **series of reads w/ no intervening writes to a variable**
    - » **cache first read in a register & eliminate the rest**
      - **oops – for a device input each read could have a different value**
      - **and you care about them all**
  - **series of writes w/no intervening reads**
    - » **no point in writing something that isn't read**
      - **eliminate all but the last write**
      - **oops – if these are device outputs then you want them all to be done**
- **Memory operation reordering**
  - **different variables map to different addresses**
  - **should be OK to reorder independent reads and writes**
    - » **last time we learned**
      - **first set PPSx then PERx (set sense and then enable)**
      - **different variables – compiler can reorder**
        - – **PERx then PPSx can be dangerous**

Bad Optimization Example

You write this code:

```
extern char MY_PTJ @ ( 0x00000268 ) ;

void Out(unsigned char data)  {
   MY_PTJ = 0;
   PTT=data;
   MY_PTJ = 1;
}
```

CodeWarrior for HCS12 gives you this:

```
     STAB   _PTT
     LDAB   #1
     STAB   MY_PTJ
     RTS
```

What is wrong?

Why did the compiler think this was OK?

Better Register Declaration

```
extern volatile char MY_PTJ @ ( 0x00000268 ) ;

void Out(unsigned char data)  {
   MY_PTJ = 0;
   PTT=data;
   MY_PTJ = 1;
}
```

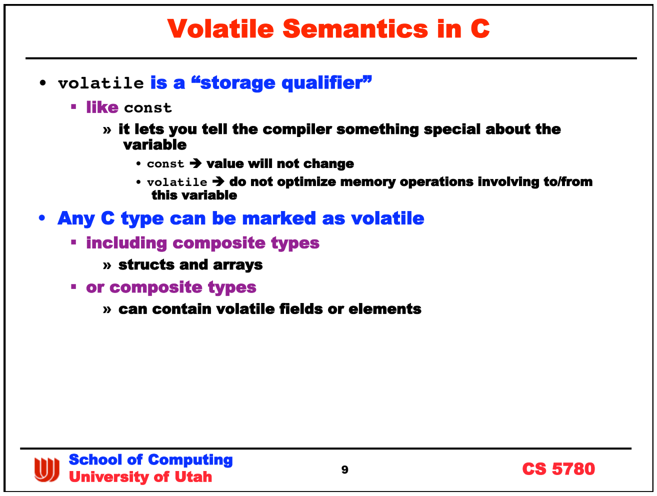
For the same C code, CodeWarrior for HCS12 gives you this:

```
     CLR    MY_PTJ
     STAB   _PTT
     LDAB   #1
     STAB   MY_PTJ
     RTS
```

Is it right now?

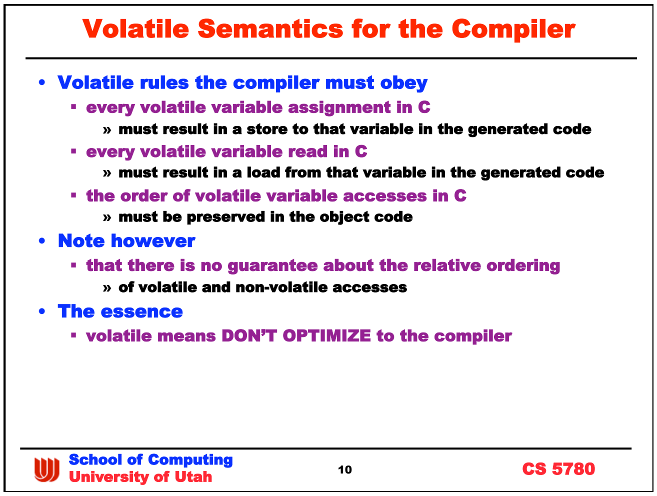
Accessing Device Registers

- **2 methods for doing it right**
  - **write assembly code**
    - » **compiler doesn't optimize this**
  - **use volatile declarations in C**
- **It's a personal choice**
  - **if you hate assembly**
    - » **then it's impossible to reliably access device registers in C without** `volatile`
    - » **it's also impossible to reliably synchronize between main and ISR routines in C code as well**
- **What this means for you**
  - **ALWAYS make a variable volatile if it:**
    - » **represents a device register**
    - » **is used to communicate with ISR's**
    - » **is used to communicate between threads**
  - **What happens if you forget?**
    - » **why?**



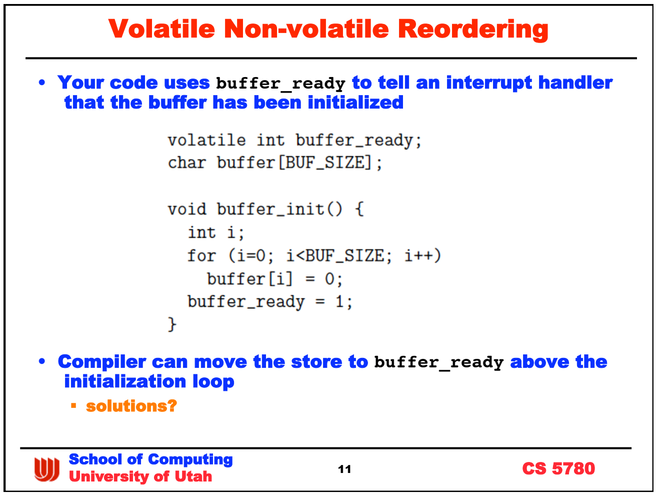
Volatile Semantics in C

- `volatile` is a "storage qualifier"
  - like `const`
    - » it lets you tell the compiler something special about the variable
      - `const` ➔ value will not change
      - `volatile` ➔ do not optimize memory operations involving to/from this variable
- Any C type can be marked as volatile
  - including composite types
    - » structs and arrays
  - or composite types
    - » can contain volatile fields or elements



Volatile Semantics for the Compiler

- Volatile rules the compiler must obey
  - every volatile variable assignment in C
    - » must result in a store to that variable in the generated code
  - every volatile variable read in C
    - » must result in a load from that variable in the generated code
  - the order of volatile variable accesses in C
    - » must be preserved in the object code
- Note however
  - that there is no guarantee about the relative ordering
    - » of volatile and non-volatile accesses
- The essence
  - volatile means DON'T OPTIMIZE to the compiler



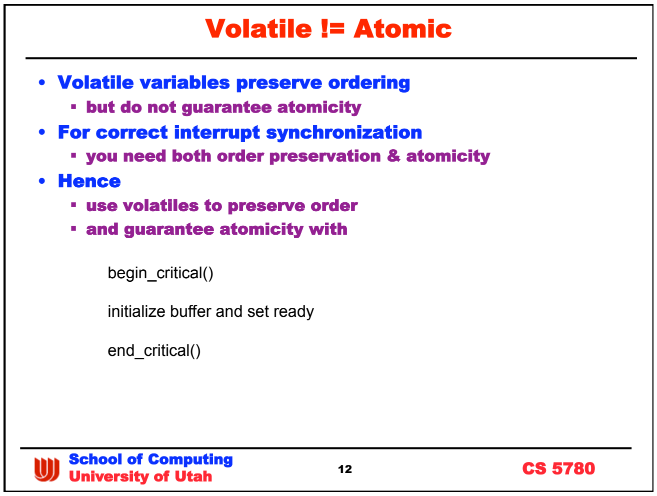
Volatile Non-volatile Reordering

- Your code uses `buffer_ready` to tell an interrupt handler that the buffer has been initialized

```
volatile int buffer_ready;
char buffer[BUF_SIZE];

void buffer_init() {
  int i;
  for (i=0; i<BUF_SIZE; i++)
    buffer[i] = 0;
  buffer_ready = 1;
}
```

- Compiler can move the store to `buffer_ready` above the initialization loop
  - solutions?



Volatile != Atomic

- Volatile variables preserve ordering
  - but do not guarantee atomicity
- For correct interrupt synchronization
  - you need both order preservation & atomicity
- Hence
  - use volatiles to preserve order
  - and guarantee atomicity with

```
begin_critical()

initialize buffer and set ready

end_critical()
```

Const Volatile

- Does this make sense?

```
const volatile int x;
```

- What does this tell the compiler

Volatile and Pointers

- You can make a pointer to a volatile int

```
int volatile *x;
```

- You can make a volatile pointer to an int

```
int *volatile x;
```

- You can make a volatile pointer to a volatile int

```
int volatile *volatile x;
```

- How do you know which of these to use?
  - even good embedded developers have to think hard about these issues
  - fall-back when these issues make you sick
    - » assembly → do it my way
    - » typedef's can help avoid some confusion

Concluding Remarks

- Belabored something that seems simple
  - why?
    - » if a large number of people have written buggy code
    - » then you might too
    - » common solution to most of these bugs was
      - treating device registers as normal variables
        - – they aren't the same
        - – I/O is all about side-effects
        - – hence order and instance preservation is important
      - hence the nerdly focus
- Bottom line
  - learn to love `volatile`

Note: midterm is a week from next Tuesday
– it would be wise to be caught up on labs & reading