CS 3410 Lab 11

Fall 2025

Cornell Bowers CIS
Computer Science

Agenda

Inline Assembly

Inline Assembly

```
__asm__ volatile(
    // Assembly instructions
    : // Output operands
    : // Input operands
    : // Clobber list
    : // Goto list
);
```

The **`volatile`** keyword instructs the compiler to avoid "optimizing your code away," so the instructions appear verbatim in the compiled program.

Inline Assembly: Operands

```c
int a_plus_3b(int a, int b) {
    int result;
    __asm__ volatile(
        "slli t0, %2, 1\n"
        "addi t0, t0, %2\n"
        "addi %0, t0, %1\n"
        : "=r" (result)
        : "r" (a), "r" (b)
        : "t0");
     return result;
 }
```

Placeholders

- `%0` refers to the first operand that appears, In this case, `result`.
- `%1` is the second operand, `a`.
- `%2` is the third operand, `b`.

Output Operand

- `=r (result)` says that the variable `result` should be placed in a register so the assembly code can write to it

Input Operand

- `"r" (a), "r" (b)` makes the arguments `a` and `b` available in registers.

Inline Assembly: Operands

```
int a_plus_3b(int a, int b) {
    int result;
    __asm__ volatile(
        "slli t0, %2, 1\n"
        "addi t0, t0, %2\n"
        "addi %0, t0, %1\n"
        : "=r" (result)
        : "r" (a), "r" (b)
        : "t0");
     return result;
 }
```

Beyond `r` and `=`, some other basic constraints and constraint modifiers are…

- `m`: The operand lives in memory.
- `f`: The operand lives in a floating point register.
- `i`: The operand is a constant integer (immediate).
- `F`: The operand is a constant floating point number.
- `+`: The operand is both read from and written to.
- `&`: The operand is written to before all (note: not any) operands have been read.

Inline Assembly: Clobber List + Goto List

```c
int a_plus_3b(int a, int b) {
    int result;
    __asm__ volatile(
        "slli t0, %2, 1\n"
        "addi t0, t0, %2\n"
        "addi %0, t0, %1\n"
        : "=r" (result)
        : "r" (a), "r" (b)
        : "t0");
     return result;
 }
```

Clobber List

- This list describes to the compiler what the assembly code (might) overwrite.
- In this case, we have `t0` in the clobber list as the assembly code write over `t0`
- This list can contain the special name `memory` to indicate that the assembly writes to memory.

Goto list

- Informs the compiler of the list of `goto` labels used in the assembly.
- We omit the `goto` list because our assembly does not use any labels.

Spinlocks

Spinlocks

A **spinlock** is a type of lock. When a thread tries to acquire a spinlock, it "spins" until the lock becomes available.

Pseudocode

```
1. Check if the shared resource is available. If it isn't, repeat step 1.

2. Try to acquire the shared resource. If unsuccessful (i.e., because another
   thread acquired it first), repeat step 1.
```

LR and SC Atomic Instructions

LR (load-reserved)

- Usage: `lr.w.aqrl rd, (rs1)`
- Loads a 32-bit word from the address in register `rs1` into register `rd`
- “Reserves” the memory address at `rs1`, tracking whether any other thread writes to this address
- A “reservation” is broken if another thread writes to this address

SC (store-conditional)

- Usage: `sc.w.aqrl rd, rs2, (rs1)`
- Attempts to write the value in `rs2` to the address in `rs1`
- Only succeeds if this thread has a reservation on this address
- If successful, writes zero to `rd`; otherwise, writes a non-zero code

LR and SC are useful for implementing spinlocks in RISC-V!

Monitors

Condition Variables

```
void print_message(int* lock, int* cond, char*
                   message, int ready) {
  spin_lock(lock);
  // Wait until the message is ready
  while (!ready) {
    wait(lock, cond);
  }
  // Print the now-ready message!
  printf("%s", message);
  spin_unlock(lock);
}
```

A **condition variable** is a concurrency mechanism that allow threads to wait for some condition to become true

- A thread waits on a condition `cond` until another thread wakes it up
- A thread can broadcast to a condition `cond` to wake up all threads that are waiting on `cond`

`futex`

In Linux, the `futex(uint32_t* uaddr, int op, ...)` system call can be used to facilitate condition variables

- `uaddr` is the address of the condition variable
- `op` is the operation to be performed by the `futex` call (e.g., `FUTEX_WAIT`, `FUTEX_WAKE`)

pthreads

pthreads

pthreads is a Unix standard library that provides implementations for thread management, synchronization, and conditioning in C

`PTHREAD_MUTEX_INITIALIZER`	Initializes `pthread_mutex_t` lock.
`PTHREAD_COND_INITIALIZER`	Initializes `pthread_cond_t` condition variable.
`pthread_mutex_lock(pthread_mutex_t lock)`	Acquire the lock `lock`.
`pthread_mutex_unlock(pthread_mutex_t lock)`	Release the lock `lock`.
`pthread_cond_wait(pthread_cond_t cond, pthread_mutex_t lock)`	Release the lock `lock` and wait on the condition variable `cond`. On return, the calling thread is guaranteed to have reacquired `lock`.
`pthread_cond_signal(pthread_cond_t cond)`	Wakes up at least one thread waiting on the condition variable `cond`.

A11 Tips

A11 Tips

- In your `rv` container, you can call `grep "#define FUTEX_" /usr/include/linux/futex.h` to find the `futex` opcodes
- More tips can be found in the A11 instructions!