Parallel Shared Memory

- Multiple processors work on a series of related jobs, numbered 0, 1, 2, ..., n
- How to coordinate assignment of work?
- How to balance load?
Static assignment

- Solution 1, divide jobs statically (k processors)
  - proc 0 does jobs 0, k, 2k, 3k, etc
  - proc 1 does jobs 1, k+1, 2k+1, 3k+1, etc
  - proc 3 does jobs 2, k+2, 2k+2, 3k+2, etc
Dynamic assignment

- Static assignment can be bad if the jobs have varied execution times – a few processors may carry most of the computational load

- Dynamic assignment
  - each processor starts with a job, same num as proc
  - job counter is a memory loc set to next job to be done
  - when a proc finishes a job, it gets the job count, increments it, and saves it back to memory, then executes that job (unless the count has reached the end of work)
Race condition

Same code on different processors
(assume $s1 holds address of job counter, $s0 the job number of this processor)

```

1a, 2a       jobstart:     lw   $s0, 0($s1)
1b, 2b       addi   $t0, $s0, 1
1c, 2c       sw    $t0, 0($s1)
1d, 2d          <start work>

...               j jobstart

```

Consider the execution sequence:
1a, 1b, 2a, 1c, 1d, 2b, 2c, 2d, .... 1z, ...

Both processors load the *same value* into $s0 and therefore do the same job.
Mutual Exclusion

- Need a method to allow only one processor to execute a block of code (the load, increment, save sequence)
- Several primitive synchronization methods are used in different systems:
  - test and set
  - register exchange
  - fetch and increment
- These must be atomic operations – no intervening instruction can occur
MIPS solution

- two coordinated instructions
  - ll  load linked
  - sc  store conditional
  - a store conditional fails if the memory location has been changed after the load-linked instruction and before the sc instruction
  - success, set source register to 1
  - failure, set source register to 0
Race Solution

Same code on different processors
(assume $s1 holds address of job counter)

1a, 2a  jobstart:  ll  $s0, 0($s1)
1b, 2b  addi  $t0, $s0, 1
1c, 2c  sc  $t0, 0($s1)
1cc, 2cc  beq  $t0, $zero, jobstart
1d, 2d  <start work>

1z, 2z  j jobstart

Consider the execution sequence:
1a, 1b, 2a, 1c, 1cc – (memory value has not been changed so no branch) – 1d, 2b, 2c, 1e, 2cc – (between 2a ll and 2c sc, instruction 1c sc has changed the memory value, so branch) – 2a, 1f, 1g, 2b, 2c, 2cc – (memory value has not been changed this time, no branch) – 2d, ... 1z, etc.
The two processors get different job values.
Fetch and Increment

The code sequence used to avoid the previous race condition is an example of a fetch-and-increment instruction implemented with `ll` and `sc`. It is not an atomic assembly instruction but it has the same effect – a processor cannot move past this sequence unless it has executed in a way logically equivalent to its being atomic.

```
1a, 2a  jobstart:  li  s0, 0(s1)
1b, 2b           addi  t0, s0, 1
1c, 2c           sc   t0, 0(s1)
1cc, 2cc         beq  t0, $zero, jobstart
```
Test-and-Set

How would you use MIPS Il and sc instructions to implement the equivalent of an atomic test-and-set?

Test-and-set atomically checks whether a memory location is set to 1, rather than 0, and at the same time sets it to 1.
Register-Exchange

Another atomic control instruction is register-exchange. For this instruction the value in a given register is exchanged with the value at the designated memory location.

How would you implement the logical equivalent of an atomic register exchange instruction using MIPS II and sc?