Prelim 1 Notes

Ambar Soni

March 21, 2014

Contents

1 What is Embedded Systems

Embedded Systems is a computer system with a dedicated function withing a larger system, oftwn with real-time computing constraints.

2 Assembly Language

The three basic part of a computation:

1. Data: values being read or produced by the computation
2. Operations: ways to manipulate data (add,sub)
3. Control: specifies what operations are to be performed on what data

2.1 RISC vs CISC

CISC

- Emphasis on hardware
- Includes multi-clock complex instructions
- Small code sizes, high cycles per second
- Includes multi-clock complex instructions
- You are limited by your slowest instruction

- Hard to pipeline since each instruction is not 1 clock cycle

RISC

- Emphasis on software
- Single-clock reduced instruction only
- Low cycles per second, large code sizes
- Smaller ISA $\rightarrow$ smaller chip
- You are limited by your slowest instruction
- Most compliers are very good at optimizing and let you program in high level

2.2 CPU and Memory

A Central Processing Unit is the hardware within a computer that carries out the instructions of a computer program by: Fetching an instruction at the pc, decoding and examining fields, updating the pc, and finally executing the operation. The CPU hardware interprets instructions (machine code) and uses gates and state elements to implement instructions. It consists of:

1. Registers: Binary N-bit, 2's complement variables for temporary storage
2. Functional Units: Given some operations + data $\xrightarrow{produce}$ results
3. Control Unit: Control logical sequence of operations

Memory: a linear array of 'registers'. Instructions are also stored in memory (von Neumann Model). Memory is one one long *spaghetti*

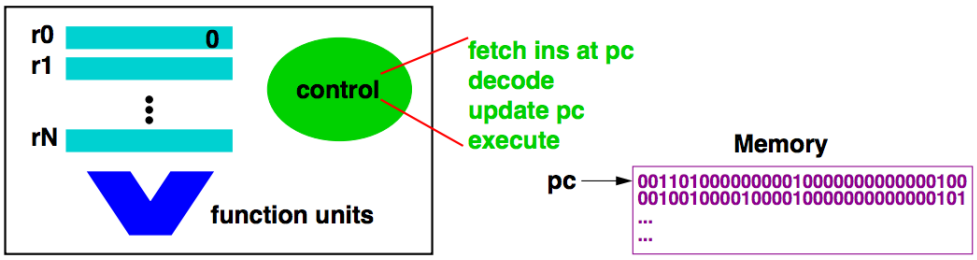


Figure 1: An Abstract Computer

2.3 Instruction Set Architecture

ISA is a contract between hardware and software: operands, data types, operations, and encoding. It specifies the language that the CPU understands. Hardware can implement however as long as software cant distinguish. Software can use any syntax as long as it translate to assembly. AKA everything is fair game people!

2.3.1 Operands

Data is either a constant (input value), in a register, or stored in memory. In the MSP 430 assembly, operands are usually 16-bit (possible to use 8-bit) and distinguished with the .b vs .w extension. For example mov.b is to move a byte sized data where as mov.w is to move a word-sized data.

Arrays are implemented using memory because remember that all memory is essentially part of the long spaghetti at the end of the day.

```
1 for (i=0; i<10; i++){
2   x = x+ a[i];
3 }
4 //a:    the location in memory
5 //a[0]: the contents stored at location (a+0)
6 //a[1]: the contents stored at either (a+1) or (a+wordsize)
```

The location of operands in specified by the *addressing mode*. The MSP 430 supports the following different addressing modes:

- *Register: `mov.w R5, R4`
- *Indexed register: `mov.w 4(R5), R4`
- Register indirect: `mov.w @R5, R4`
- Indirect auto-increment: `mov.w @R5+, R4`
- *pc-relative label: `mov.w label,R4`
- *Absolute label: `mov.w &label,R4`
- Immediate: `mov.w #4,R4`

2.3.2 Endianness

Endianess comes into play when you want to break a large value (such as words) into several small ones (such as bytes). For implementation, there are two approaches to the order of the smaller parts.

1. Big Endian: Store the most significant byte in the smallest address
2. Little Endian: Store the least significant byte in the smallest address

For example, if we wanted to store the value xFEED into a memory array, we could use the following:

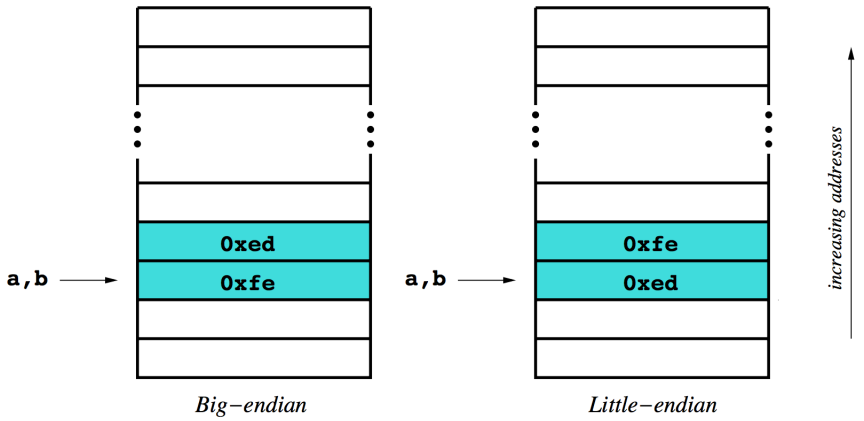
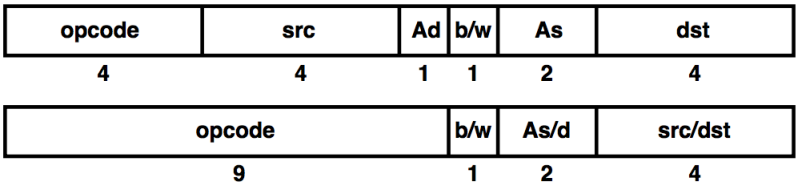


Figure 2: Big Endian vs Little Endian

3 Most Commonly Used Instructions

add(.b)	src,dst	dst←src+dst	all flags
addc(.b)	src,dst	dst←src+dst+C	all flags
cmp(.b)	src,dst	dst-src	all flags
dadd(.b)	src,dst	dst←src+dst+C (dec)	all flags
mov(.b)	src,dst	dst←src	no flags
rra(.b)	dst	(dst,C)←dst>>1	all flags
rrc(.b)	dst	(dst,C)←(C,dst)>>1	all flags
sub(.b)	src,dst	dst←dst+$\overline{\text{src}}$+1	all flags
subc(.b)	src,dst	dst←dst+$\overline{\text{src}}$+C	all flags
sxt	dst	sign extend byte	no flags

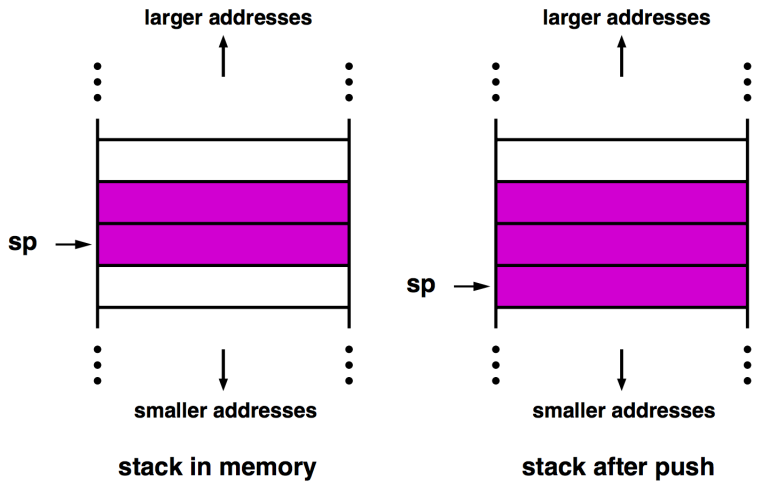


4 Function Calls

Functions are self contained modules of code that accomplish a specific task. In the MSP 430, there are general guidelines for implementing and using functions that ensure that your data is saved and the you get the result you desire.

4.1 Stack

The stack is a LIFO data structure that maintains the invariant that the last element inserted is the first element to be removed. This is an ideal structure for function calls and especially for recursion. In the MSP 430, it looks something like this:



To push to the stack we use the following command:

```
1  SP ← SP-2
2  @SP ← src
```

4.2 Stack Frame

This is not the same as a stack! A stack frame is the part of the stack that contains the infromation revelvant to that call of the function. This usually involves the function call's parameters, return address, and local variables.

4.3 Order of Operations

To properly implement a function call the following convention is obeyed:

1. User pushes the caller saved registers onto the stack (R11-R15)
2. User pushes any parameters onto the stack
3. User uses the keyword *call* to push the return addr onto the stack and jump to the function
4. Function pushes callee saved registers onto stack (R4-R10)
5. Function pushed any local variables on to the stack
6. Function uses key word *ret* to put the return addr into the PC and return to the caller. Its a pseudo instruction for mov.w @R1+, R0

Therefore, the stack should look something like this:

R11
R12-R15
parameters
ret address
R4-R10
local vars

5 Assembling the Program

5.1 Assembler

The assembler translates assembly language programs into machine code. It expands pseudo operations, converts assembly to machine code, and outputs an object file. Additionally, they keep track of where the jumps are, and where references to labels are. There are two approaches to assemblers:

1. Two-Pass: It allocates instruction and determines addresses on first pass and then assembles instructions knowing all labels.
2. One-Pass: It assembles instructions putting zeros for unknown offsets and keeps track of unfinished instructions. When labels appear or at the end of oass one, it fills in unfinished instructions.

5.2 Linker

The linker takes a collection of object files and libraries and generates an exectuable program.

6 Input Output

Three common ways to implement I/O are:

- Special Instructions: They access the IO ports and the code is easy to follow but you give up one opcode.
- Special Registers: Some registers are mapped to I/O ports but there are only a limited number of registers anyways.
- Memory Mapped I/O: Certain memory address are linked to I/O ports and this saves new instructions since we can just use load and store (mov.w).

6.1 Polling

Polling is a common way to handle when to read input. Implementing polling-based input requires a loop that checks the value of a certain input register until its value is set by some hardware. Once we read that the input flag is set, we can read and handle the input value. While the process is polling the input, the CPU cannot execute any other instructions. However, polling is very simple to implement and has very low overhead (3 instruction loop) and can often be faster than interrupt driven input.

6.2 Interrupts (async)

In interrupt-based input, the CPU can execute other instructions and when the hardware of the input is set, it sends a signal to the CPU that there is an input to be read. At this point, the current process is interrupted, its current PC and SR are pushed to the stack, the corresponding ISR is run, and finally the PC and SR are popped off so that the process can resume where it was before. All registers become callee saved registers because it impossible to tell when an interrupt is invoked. At the end we use to command *reti*. The SR has a bit that says if the CPU can be interrupted or not. Interrupt driven input also requires more hardware support than polling. The asynchronous tag means that we cannot at which instruction the interrupt occured.

6.3 Exceptions (sync)

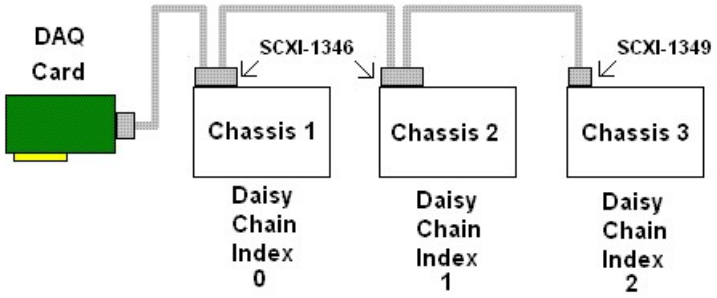
Exceptions are also handled through interrupts, however they are synchronous with the execution of the program (divide by 0, page fault). For example a *Page Fault* occurs when a page needed is in DISC so the ISR will bring the page into memory and then return back to the same instruction. The synchronous tag means that you can pin point exactly which instruction caused the exception or trap.

6.4 Service Routines (TRAPS)

Service routines are responsible for reading and writing to the DISC. They encapsulate simple tasks for the user and invoke changes of privilege. For example, when you boot up, you are in Super User mode so that you can define where SVC are even located and then you change back to user mode. It is a particular instruction.

6.5 Daisy Chaining

Daisy Chaining is a wiring scheme in which multiple devices are wired together in sequence and they share the *same* port.



The closest connection has the highest priority. The problem is if there is any electrical disconnect, all priorities will be lost.

6.6 Interrupt Vector

- The Interrupt vector deallocates the system call 'name' from actual ISR address.

- It allows flexibility in designing ISR code for a particular service
- Provides an organized 'menu' of services

7 Concurrency

Concurrency is defined as running in parallel or operating at the same time. There are two types or resources involved here:

- Shared Variables: accessed by multiple processes and their single read and write is atomic. Must be stored in global memory and not stack.
- Private Variables: accessed by only 1 process and are not interfered

`Concerns in Mutual Exclusion:`

- Safety: At any moment, at most one process is in the Critical Section
- Fairness: If a process is continuously enabled, it will eventually get a chance to execute (Weak Fairness). Given many chances to execute, all processes will have a chance of running (Probablistic Fairness)
- Progress: There is no Deadlock or Livelock

7.1 Turn Approach

```
P1 : while (1) {
       NCS1;
       while (turn!=1) ;
       CS1;
       turn = 2;
     }

P2 : while (1) {
       NCS2;
       while (turn!=2) ;
       CS2;
       turn = 1;
     }
```

- Safety: Yes - only one CS will execute at a time
- Fairness: Yes - probablistic fairness as long as there is progress but not weak fairness if crash in NCS
- Progress: No - if you crash in one CS, no process can run

7.2 Dekker's Algorithm V1

```
P1 : while (1) {
       NCS1;
       while (x2);
       x1=1;
       CS1;
       x1=0;
     }
```

```
P2 : while (1) {
       NCS2;
       while (x1);
       x2=1;
       CS2;
       x2=0;
     }
```

- Safety: No - if x1 and x2 = 0, both can access CS at same time
- Fairness: Yes - probablistic fairness, not always weak fairness
- Progress: Yes - if one CS crashes, other process can still run

7.3 Dekker's Algorithm V2

```
P1 : while (1) {
       NCS1;
       x1=1;
       while (x2) {
         x1=0;
         x1=1;
     }
       CS1;
       x1=0;
     }
```

```
P2 : while (1) {
       NCS2;
       x2=1;
       while (x1) {
         x2=0;
         x2=1;
     }
       CS2;
       x2=0;
     }
```

- Safety: Yes - only one CS will execute at a time
- Fairness: Yes - after one process completes CS, it yields to other and one NCS crash doesn't affect other
- Progress: No - livelock can occur since x1 and x2 start as 1 and wait for each other to become 0. Classic 'after you, no after you' problem

7.4 Dekker's Algorithm Final

```
P1 :  while (1) {                P2 :  while (1) {
        NCS1;                            NCS2;
        x1=1;                            x2=1;
        while (x2) {                     while (x1) {
          x1=0;                            x2=0;
          while (turn!=1) ;                while (turn!=2) ;
          x1=1;                            x2=1;
        }                                }
        CS1;                             CS2;
        x1=0;turn=2;                     x2=0;turn=1;
      }                                }
```

- Safety: Yes - only one CS will execute at a time
- Fairness: Yes - after one process completes CS, it yields to other and one NCS crash doesn't affect other
- Progress: Yes - with the use of turn, there is no chance of deadlock of livelock

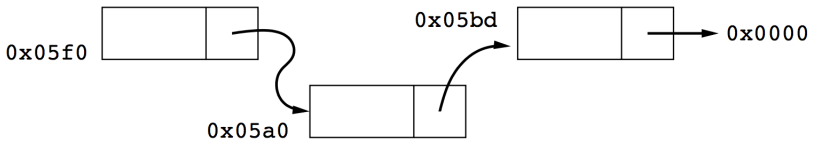
8 Time Sharing - Implementing Concurrency

A process can either be in the *ready*, *blocked*, or *running* state of a process contains:

- Code: PC
- Data: Registers, memory variables (private and shared), stack
- State: Flags (SR)

Each process contains its own stack space and has a SP. The rest of the state is *saved on the stack*. We maintain a queue of process in the ready state:

```
1 struct process_state {
2   uint sp;
3   struct process_state *next;
4 };
5
6 typedef struct process_state process_t;
```



8.1 Context Switching

Context Switching gives the illusiong of concurrency even though all the processes are sharing one CPU. Timer interrupts are used to stop process execution. In the handler, we pick up the next ready process from the queue and update registers to correspond to that process. The timer sets the quantum of the execution (*time slice*). The process of saving and restoring process state is called a context switch. The procedure for switching processes using the Round Robin method:

1. Timer triggers interrupt
2. Push the PC, SR, and *all* registers of current process
3. Update the SP of the current process
4. Grab the first process on the ready queue and make it the current process
5. Restore the Registers in reverse order, and use *reti* to restore SR and PC

9 Locks

A lock supports two basic - atomic operations:

- lock(l)
- unlock(l)

Note this is simple an abstraction and locks can be implemented in many different ways.

```
P1 : while (1) {
       NCS1;
       lock(l);
       CS1;
       unlock(l);
     }
```

```
P2 : while (1) {
       NCS2;
       lock(l);
       CS2;
       unlock(l);
     }
```

Strictly speaking, simple locks are not fair because there is no guarantee that any one of the processes competing will ever win the lock in a finite time.

9.1 Test and Set

Test and set instruction is an instruction used to write to a memory location and return its old value as a single atomic operation. It sets the Z flag according to the value read and sets the lock's value to 0. Below is how we implement the lock() function seen above:

```
Lock:   t&s &l
        jz Lock
        ret

Unlock: mov.w #1, l
        ret
```

The advantage of this implementation is that it maintains an atomic instruction for RMW. It is very efficient and has low overhead. However, it is still a spin lock and 1 quantum is wasted just looping on a variable.

9.2 Blocking Locks

To avoid using a spin lock, we can use a blocking primitives. The basic idea behind this is that the OS either gives the process access to the CS or it kicks it out of the CPU (into blocked queue). However, there is more overhead here because we have to switch to the SU mode. To implement this, we define a Trap routine to handle the locking and this is useful because traps have access to OS functions as we are in the SU mode. This is good for very small critical sections since we have to disable all interrupts. The pseudo code for this is as follows:

```
GIE = 0; //OS does this internally, acts as a lock as all interrupts are
    disabled
if(l == 1){
  l = 0;
  return;
}
else{
  move process from ready queue to blocked queue;
}
```

9.3 Readers and Writers Example

There are two types of processes. Readers read a shared variable and Writers modify a shared variable. Reads and Writes are mutually exclusive. A simple approach uses two shared variables:

- nw: number of writers
- nr: number of readers

```
enter_r:  lock(m);          enter_w:  lock(m);
          while (nw) {                while (nw∨nr>0) {
            unlock(m);                  unlock(m);
            while (nw);                 while (nw∨nr>0);
            lock(m);                    lock(m);
          }                           }
          nr=nr+1;                    nw=1;
          unlock(m);                  unlock(m);
```

Note: the locks in this case helps to implement a higher level construct of readers and writers mutual exclusion and does not guarantee something for the Critical Section. The Critical Section is protected by *nr* and *nw* and these two shared variables are protected by lock/unlock. We must also implement an exit() function so that the readers eventually allow writers to write. Also, the while loops in this example indicate that there is a spin lock which can be unwanted and in the next section we will see another approach to this problem that avoids the spin lock.

10 Higher Level Constructs

10.1 Condition Variables

A condition variable indicates an event and has no value. More precisely, one cannot store a value into nor retrieve a value from a condition variable. If a thread must wait for an event to occur, that thread waits on the corresponding condition variable. If another thread causes an event to occur, that thread simply signals the corresponding condition variable. Thus, a condition variable has a queue for those threads that are waiting the corresponding event to occur to wait on. Each condition variable has two basic operations: *wait(l,c) and signal(l,c)* where wait is a blocking operation.

wait(l,c):

- Waits for a condition to be signaled
- While it is waiting, the lock is released
- When we continue after the wait, the lock has been re-acquired

signal(l,c):

- Signals the condition
- It also releases the lock so that the next use of the release lock is a process that was previously blocked on the wait

waiting(l,c):

- Returns true or false depending on whether or not there is a process blocked on the condition - you must hold the lock when this is executed.

10.2 Another Look at Readers and Writers

We can use Condition Variables to implement the read and write functions which eliminates spin locks.

```
enter_r:  lock(m);
          if (nw) {
            wait(m,cr);
          }
          nr=nr+1;
          unlock(m);
```

```
exit_r:   lock(m);
          nr=nr-1;
          if (nr==0∧waiting(m,cw)) {
            signal(m,cw);
          }
```

However, when we don't to serialize readers, especially when there are no writers present. To let ready readers execute, we can change *enter_r* so that there are cascaded readers:

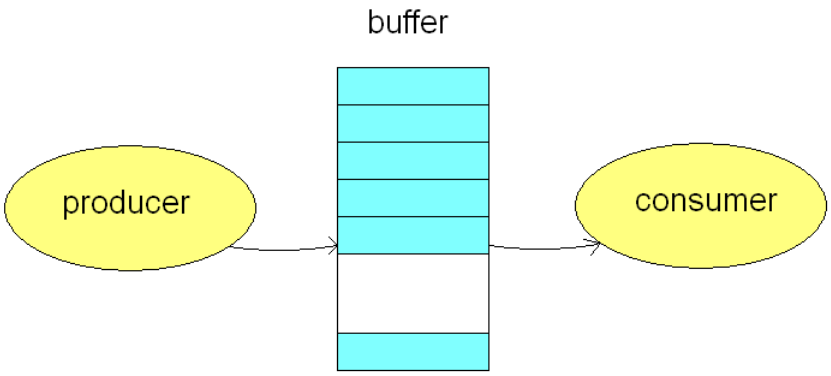
```
enter_r:  lock(m);
          if (nw) {
            wait(m,cr);
          }
          nr=nr+1;
          if (waiting(m,cr)) {
            signal(m,cr);
          } else {
            unlock(m);
          }
```

10.3 Semaphores

Think of semaphores as bouncers at a nightclub. There are a dedicated number of people that are allowed in the club at once. If the club is full no one is allowed to enter, but as soon

as one person leaves another person might enter. It's simply a way to limit the number of consumers for a specific resource.

For example, a semaphore can be used to solve the Producers and Consumers problem where we have processes that constantly try to write to a buffer and processes that constantly read from it.



A very simple implementation of the following is shown below and we can see that there is spin lock that can be avoided:

```
1  typedef struct{
2      lock_t lock;
3      int count;
4  }sem_t;
5
6  //Initilialze semaphor with count as the limit on the resource
7  void intit(sem_t *s, int c){
8      lock(s->lock);
9      s->count = c;
10     unlock(s->lock);
11 }
12
13 //Write: Adds 1 to the count
14 void v(sem_t *s){
15     lock(s->lock);
16     s->count++;
17     unlock(s->lock);
18 }
19
20 //Read: Waits until atleast 1 resource, and then decrements count
21 void p(sem_t *s){
22     lock(s->lock);
23     while(s->count < 1){
24         unlock(s->lock);
25         while(!s->count);
26         lock(s->lock);
27     }
```

```
    s->count--;
    unlock(s->lock);
}
```

To improve performance and avoid the spin lock, we can use condition variables so that other processes can utilize the CPU while a reader is waiting for at least one resource to be available.

```
void v(sem_t *s){
    lock(s->lock);
    s->count++;
    if(waiting(s->lock,s->count > 0)){
        signal(s->lock,s->count > 0);
    }
    unlock(s->lock);
}

void p(sem_t *s){
    lock(s->lock);
    if(!s->count){
        wait(s->lock, s->count > 0);
    }
    s->count--;
    unlock(s->lock);
}
```

11 Event Driven Programming and Low Power Modes

When the CPU is not doing anything, it goes to sleep → low power mode. At this point, the processor waits to be awaken by an interrupt and this does require hardware support. The MSP430 has a 32 kHz crystal that drives all the low power peripherals (sensors). This same crystal is fed into Digital Clock Signal that produces a higher frequency that the CPU and high power peripherals use.

```
while(1){
  wait for any interrupt;
  if(interrupt 0){
    operation 0;
  }else if(interrupt 1){
    operation 1;
  }...
}
```

The special codes to indicate low power modes are stored in the Status Register. When the processor goes into low power mode, its SR is modified and it goes to sleep. When an interrupt occurs, the SR is pushed into the stack and a cleared SR is given to the ISR to work with. When we return from the ISR, we don't want to restore the SR because that would mean we go back to low power mode. Instead, the ISR uses a hack where it goes to the stack and modifies the SR before *reti* is executed.

12 Extras

12.1 Peterson's Algorithm

Peterson's algorithm is a concurrent programming algorithm for mutual exclusion similar to Dekker's algo that allows 2 processes to share a resource without conflict. it uses two shared variables: *flag* and *turn*. A *flag[n]* value of *true* indicates that the process n wants to enter the CS. Entrance to the CS is granted for the process P0 if P1 does not want to enter its critical section or it P1 has given priority to P0 by setting *turn* to 0.

```
1  bool flag[0]   = false;
2  bool flag[1]   = false;
3  int turn;
4
5  P0:      flag[0] = true;
6  P0_gate: turn = 1;
7           while (flag[1] && turn == 1)
8           {
9               // busy wait
10          }
11          // critical section
12          ...
13          // end of critical section
14          flag[0] = false;
15
16 P1:      flag[1] = true;
17 P1_gate: turn = 0;
18          while (flag[0] && turn == 0)
19          {
20              // busy wait
21          }
22          // critical section
23          ...
24          // end of critical section
25          flag[1] = false;
```

- Safety: Yes - P0 and P1 can never be in the critical section at the same time: If P0 is in its critical section, then flag[0] is true. In addition, either flag[1] is false (meaning P1 has left its critical section), or turn is 0 (meaning P1 is just now trying to enter the critical section, but graciously waiting), or P1 is at label P1_gate (trying to enter its critical section, after setting flag[1] to true but before setting turn to 0 and busy waiting)
- Fairness: Yes - there is weak fairness, both processes will eventually get CS and if you crash in NCS, the other process can still execute
- Progress: Yes - A process cannot immediately re-enter the critical section if the other process has set its flag to say that it would like to enter its critical section and there is no chance for livelock or deadlock.