Operating System and Virtual Memory

Computer Science 240

Laboratory 12

In lecture, you have introduced to the idea of an *operating system,* system software that manages the resources of a computer for multiple programs. Along with this, you are learning about how virtual memory can be used to allow efficient and safe sharing of memory among multiple programs.

Machines have instructions for handling operating system tasks which are not accessible by the average user of the system. These instructions are similar to the Mini-MIPS instructions we have learned about in the sense that there is underlying circuitry in the processor which fetches, decodes, and executes the instructions to perform whatever task is required. Therefore, to add an operating system, it is necessary to add instructions/hardware to the basic data and control path of the machine.

Although it is beyond the scope of what we can accomplish in lab to completely implement virtual memory or an operating system for our lab machine, today in lab you will experiement with a few circuits which should enable you to understand the basic idea of how these techniques are implemented in hardware.

Operating System

A primitive operating system has been added to Mini-MIPS in *os_datapath.cct,* the third circuit from today's lab materials:

This circuit can execute an operating system loop and service 3 processes. Each process is allowed to execute for 16 instructions before it is moved to the tail of the queue.

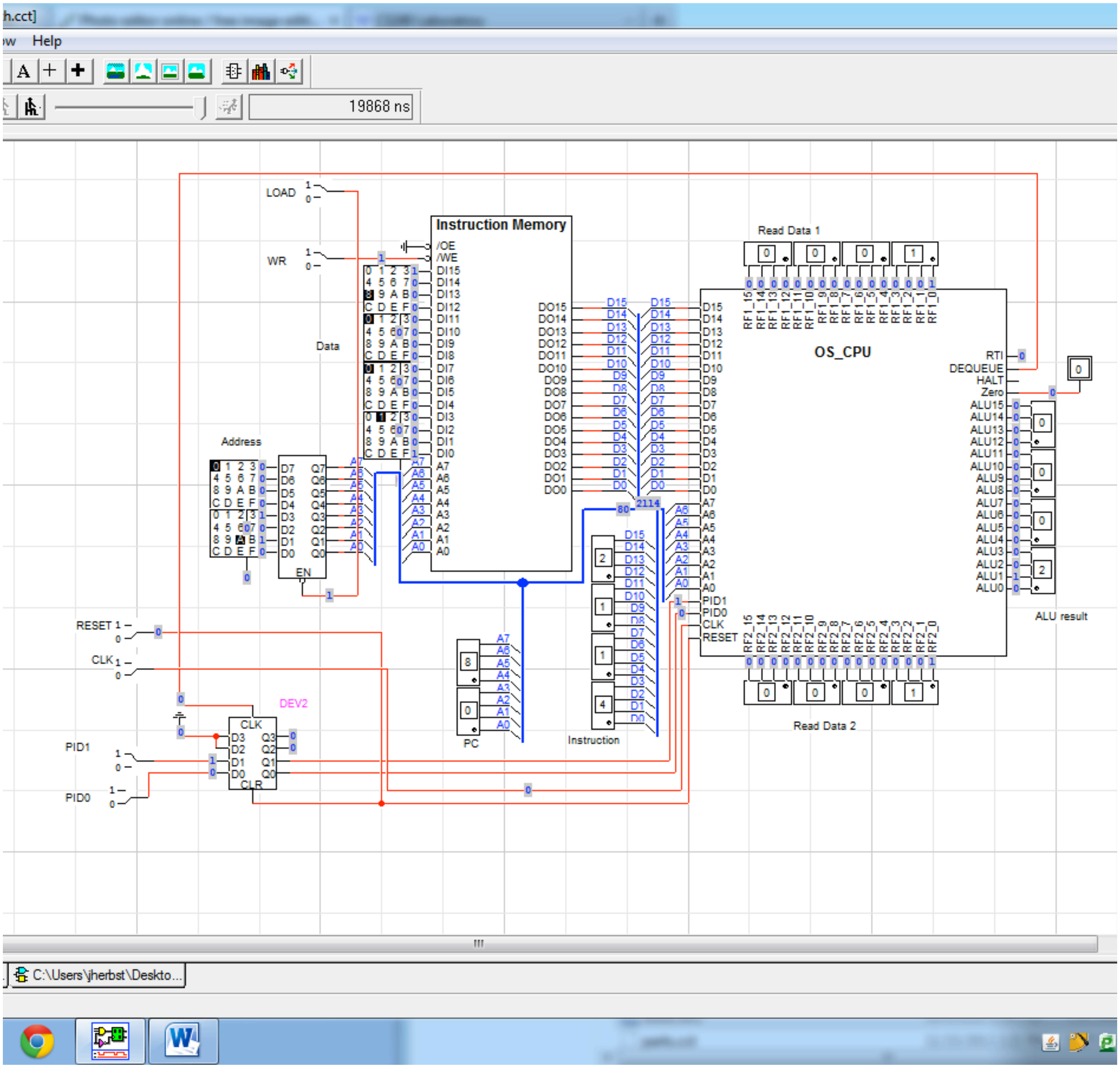
In order to reduce the complications of using virtual memory with Mini-MIPS, the *os_datapath.cct* separates the existing 256 word memory into 4 quadrants, and assigns the operating system and each of the processes 64 addresses. The operating system always uses address 0 – 63, PID 1 uses 64 – 127, PID 2 uses 128 – 191, and PID 3 uses 191 – 255. The appropriate section of memory is accessed by using the PID as the two most significant bits of the address throughout the datapath.

The PC and the register file have been modified so that the state is saved in duplicate registers. The appropriate devices are selected by the PID.

Three new instructions are needed for implementing a primitive operating system with Mini-MIPS*: HALT, DEQUEUE*, and *RTI.* None of these instructions require operands.

> **HALT** is actually a user instruction, but is an important addition to the basic instruction set, because it allows a graceful exit from a program.
>
> **DEQUEUE** accepts a new PID from the process queue into the processor and stores the PID at address 0 in data memory.
>
> **RTI** transfers control of the processor (PC, registers, and memory) to the process, through use of the PID.

On the next page is a screenshot of the circuit. It looks quite similar to the Mini-MIPS single cycle implementation. For simplicity, we simulate the PIDs rather than using a queue.


h.cct]
w Help
19868 ns
LOAD
WR
Data
Address
Instruction Memory
/OE
/WE
EN
Read Data 1
OS_CPU
RTI
DEQUEUE
HALT
Zero
ALU result
Read Data 2
PC
Instruction
RESET
CLK
PID1
PID0
DEV2
CLK
CLR
C:\Users\jherbst\Deskto...

You may recall the 9 basic Mini-MIPS instructions:

Opcode	Instruction	Operands
0000	LW	Rs Rt offset
0001	SW	Rs Rt offset
0010	ADD	Rs Rt Rd
0011	SUB	Rs Rt Rd
0100	AND	Rs Rt Rd
0101	OR	Rs Rt Rd
0110	SLT	Rs Rt Rd
0111	BEQ	Rs Rt offset
1000	JMP	12-bit offset

It is important to note that JMP uses all 12 remaining bits as an offset to an absolute address, but since addresses can be no larger than 256, the offset can not exceed 128, which can be expressed in 7 bits. That means the top 4 bits of the offset can never be non-zero. So, those bits can be used to encode the additional instructions.

The new instructions are also going to have an opcode of 1000, but the next 4 bits will be used to distinguish JMP from the new instruction (much the way regular MIPs uses an extra field to specifiy certain operations).

1000 0000	JMP
1000 1000	DEQUEUE
1000 1100	RTI
1000 1111	HALT

One of the clever things about this scheme is that it leaves lots of room for other instructions to be added, as well (since this is truly a minimal system; it would be nice to have I/O as well as the operating system commands that would be necessary to handle the multi-user access).

Also, choosing the JMP opcode for the new instructions does not require much modification of the data and control paths, because the JMP instruction does not use the regular datapath.

Operating System Code
The operating system is a simple loop:

```
Addr    Instruction
00      JMP 0 0 1      # loop actually starts at addr 2,
02      DEQUEUE        # get current PID and stores in address 0 of data memory
04      LW R0 R2 0     # test PID to see if it is non-zero
06      BEQ R0 R2 0    # if 0,  no processes in queue, keep looping until one shows up
08      RTI            # transfer control of the processor to the process, using the PID
0A      JMP 0 0 1      # repeat the loop when the process has executed for 16 clock cycles
```

Exercise 3: Open the *os_datapath.cct* file, and examine the underlying circuit of the *OS_CPU* device (there should be tabs at the bottom which you can select for the sub-circuits; if not, you can double-click on the device to open the sub-circuit in a separate window).

1. Find *HALT, DEQUEUE,* and *RTI* . Explain how those instructions are recognized in the hardware:
The highest 8 bits of the instruction are decoded to recognize the instruction (i.e. if the most significant bit is 1 and the next 4 bits are 1111, it is a HALT instruction). This can be accomplished with a simple AND gate.

2. Find PID1 and PID0, and take note of changes in the configuration of the PC and Regfile. What extra inputs are there to the devices, and how do you think they are used? The regfile now takes as inputs two select lines, which are derived from the PID. The select lines are used to choose a set of registers for use with each process. Similarly, the PC takes the PID bits, in addition to DEQUEUE and RTI, in order to select the PC value for the process being serviced.

3. The processes have the following code (these programs do not do anything useful, but they are different from one another in number and type of instructions so that you can see different programs are running).

Process 1

Addr	**Instruction**
40	5202
42	5333
44	5444
46	5555
48	5555
4A	5555
4C	8000

Process 2

Addr	**Instruction**
80	2114
82	2448
84	8000

Process 3

Addr	**Instruction**
C0	2146
C2	3448
C4	6018
C6	7480
C8	7111
CA	7220
CC	8000

4. Run the circuit; begin by setting RESET = 1 and back to 0. Set PID1 = 0 and PID0 = 1.

   This initializes all the registers to 0, and you begin execution at address 0 (which is the first instruction of the operating system). The PID is set to process 1.

   You should see: Addr = 00 Data = 8001

5. Set CK to 1 and back to 1. Each time, observe the address and data. You should step through addresses 0 – 6, and then jump to address 40 as soon as the RTI at address 8 is sensed.

6. Execute 16 instructions of process 1 by toggling CK 16 times.

   You should see the operating system resume execution at address A. Take note of the last address executed for process 1 (you should return to this address next time you execute this process).

7. Set PID1 = 1 and PID0 = 0 (this sets the PID to process 2).

8. Repeat steps 5 and 6. You should see 16 instructions of process 2 execute, and then back to the operating system.

9. Set PID 1 = 0 and PID0 = 1 again (back to process 1).

10. Repeat steps 5 and 6. Notice that you begin executing process 1 at the point where you stopped earlier after 16 cycles.

11. This time back at the operating system, set PID1 = 1 and PID0 = 1 (process 3).

12. Repeat steps 5 and 6. Once again, you should see 16 instructions of process 3 execute, then back to the operating system.

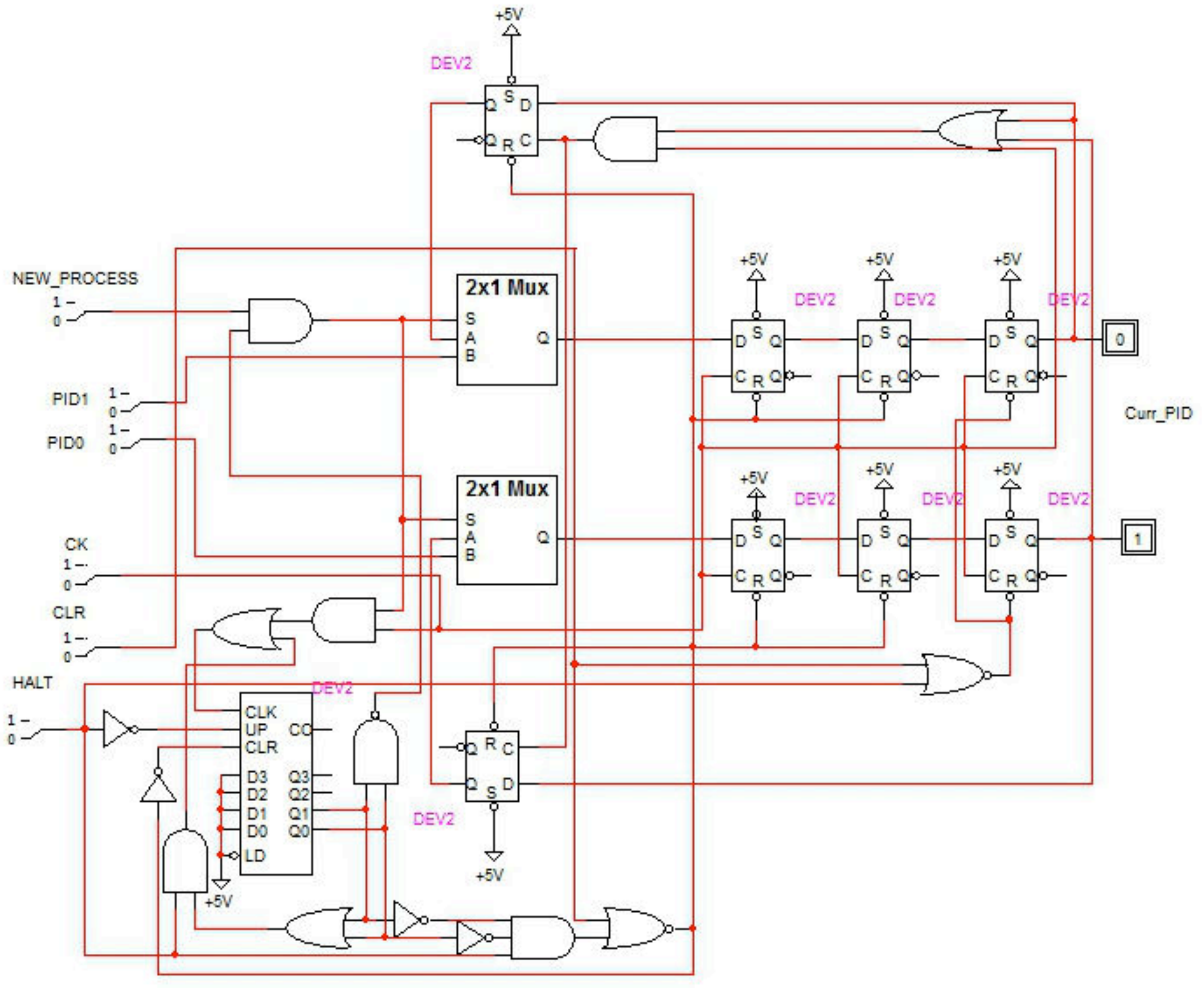
Process ID Queue
In order to handle multiple users, an operating system must have a strategy for accepting and scheduling processes. One way to do this is to assign an ID or number to each process, and then maintain a *queue* of the process IDs, so that the processes can be serviced in some order.

The circuit *queue.cct* which is part of the lab materials for today, allows up to three processes to be scheduled.

A process ID (PID) is a 2-bit value, which can by 00 (if operating system code is being executed), or 01,10, or 11 (to correspond to the three processes). The current PID (at the *head* of the queue) is displayed on the binary probes to the right of the circuit. In the diagram, the current PID = 01, and is stored in the rightmost pair/column of D flip-flops.

The PID of the next processes scheduled are stored in the other two pairs/columns of D flip-flops to the left.

When the current PID is *dequeued***,** or passed to the processor, the next PID moves forward to take its place. However, the current PID is also fed back in to the tail of the queue, so that it will have another turn to be processed. The tail of the queue can get a new value from either a new PID (if there are fewer than 3 in the queue), or from the current PID. If the queue is full, it will act as a circular shifter until one or more of the processes *halt* (complete execution).



Exercise 2: Operate the circuit by doing the following:

1. Set CLR=1 and back to 0.

   This corresponds to restarting the system, at which point the operating system has control. The 3 PIDs are set to 0.

2. Set PID1 = 0 and PID0 = 1 (PID = 01).

   The first PID in the queue will be 01.

3. Set NEW_PROCESS = 1 amd set CK to 1 and back to 0. Set NEW_PROCESS = 0
   Examine the value at the outputs of the leftmost column of flip-flops to see the value of 01
   NOTE: LogicWorks has a **Show Values** option in the Simulation Menu which should allow you to observe values more easily.

4. Repeat steps 2 – 3 for PID 10 and 11.

5. You should now to able to observe the three values. If you continue to clock the circuit, you can see how the values circulate from the head to the tail of the queue.

6. Set HALT = 1 and back to 0.

   This simulates a process that has completed execution, and the current PID should be set to 0.

   Clock the circuit several times. It is possible that the PID will show up again. If it does, you should set HALT = 1 and back to 0 again. Continue to clock until the PID no longer shows up.

   You can then do this for the other PIDs until the queue is empty and has all 0's again. This corresponds to the operating system sitting in the schedule loop and waiting for requests.

7. Examine the circuit and explain the purpose of the multiplexers and the extra set of D flip-flops which feed the multiplexors. The next PID which goes to the tail of the queue is either a new process requesting service, or the current process (which goes to the tail after being serviced). The MUX selects a new PID if it is being applied, and stores the current PID for one cycle, so that the current PID will go to the tail of the queue on the next cycle after the new PID has entered the queue.

8. This circuit uses a counter device:
   LD stands for Load, which means initialize the Q outputs to the values of the D inputs
   CLR means initialze the outputs Q0..7 to all 0's
   EN stands for Enable, the device will not count/increment if EN is not active
   UP = 1 means increment the Q outputs each time the CLK is toggled.

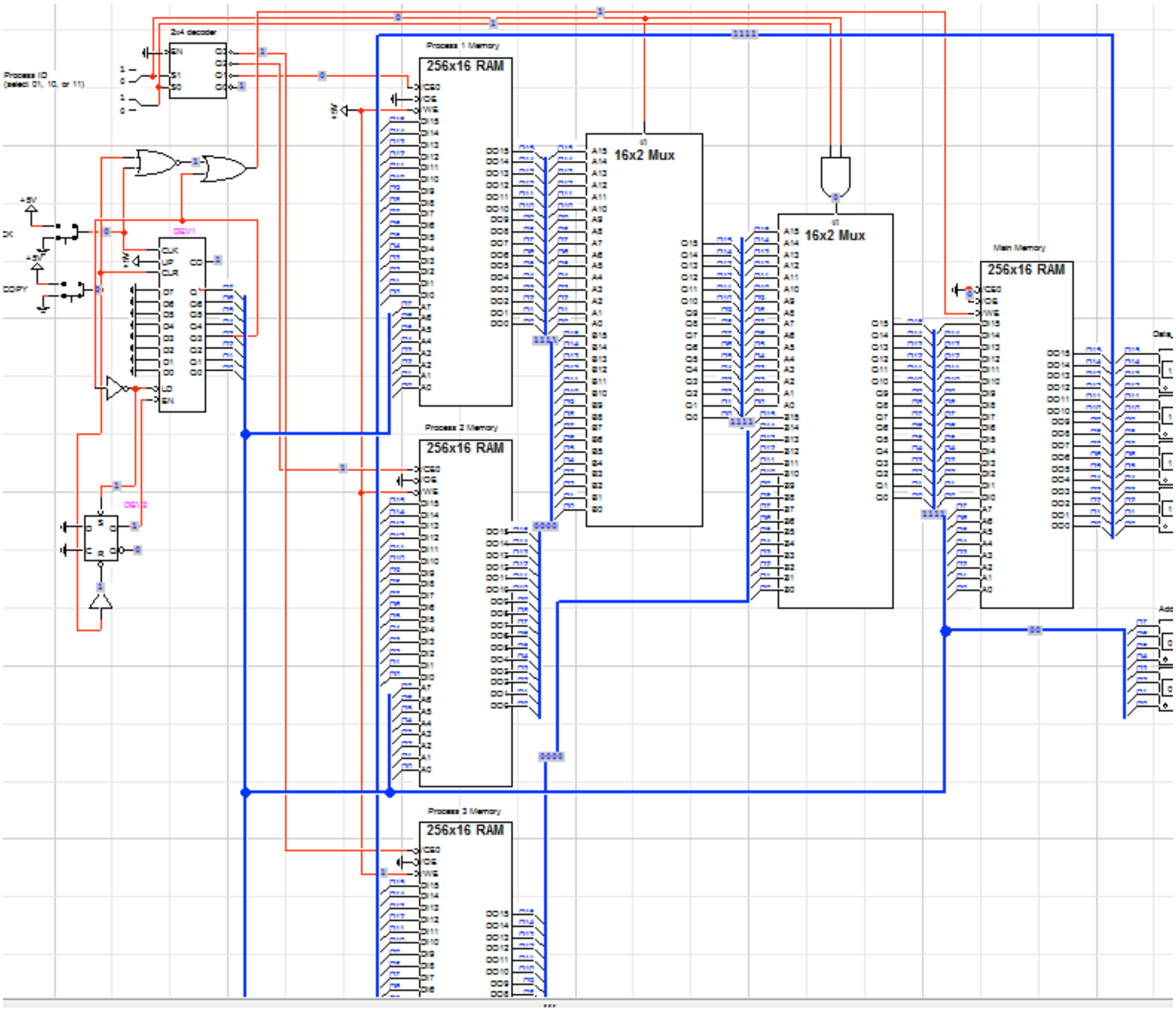
Describe the purpose of the counter: The counter keeps track of how many processes are in the queue (0 – 3). It decrements the counter if a process ends, and increments it when one is added.

Virtual Memory
Main memory in any computer is not large enough to hold all the programs at once that you (and other users) might like to run at any given time. It is often necessary to bring programs in from other storage devices (cache memory, hard drive, flash drive, etc.) into main memory when it is time to run a program.

Exercise 3: Open the file *virtual_memory.cct* that you received as part of the lab materials for today. This circuit contains four memory devices. Three of the device can be thought of as the memories for three *processes* (programs) which are being executed by a single Mini-MIPS processor. The fourth memory is the *main memory*, which is the device which the processor will actually use during execution.

The circuit will select a memory device based on the *process id*, or PID (which can be 1, 2, or 3), and will copy the contents of the device to the main memory, so that the process can be executed.

For simplicity, only 8 locations are copied from each device (although this could easily be extended to include the entire address space).



The contents of the first 8 memory locations in the Processor 1 Memory is

Address	**Data**
00	1111
01	1112
02	1113
03	1114
04	1115
05	1116
06	1117
07	1118

To copy this data to the main memory, do the following:

1. Set PID to 01
2. Click on the Copy switch (it is a pushbutton switch, so click once to toggle high then low)
3. For each address, click CK (also a pushbutton switch) to copy the contents of the current address in the processor memory to the main memory. This also advances the address to the next location.

When the 8 address locations have been copied, the address will return to the starting location of 00, and continuing to click CK will have no effect (the circuit can detect when all 8 locations have been copied).

Repeat the process for Processor 2 and Processor 3, and record the contents of the devices in the table below:

Processor 2 Memory

Address	Data
00	2262
01	2272
02	2282
03	2292
04	22A2
05	22B2
06	22C2
07	22D2

Processor 3 Memory

Address	Data
00	3330
01	3334
02	3338
03	333C
04	3331
05	3335
06	3339
07	333D

Examine the circuit and answer the following questions:

1. What are the multiplexors used for? How is the decoder involved in that operation?

The multiplexers select 1 of the 3 memories to copy to the main memory. The output of the decoder, which has one active line for each of the PIDs, is used as the select inputs to the multiplexers.

2. The devices on the left labeled *DEV1* and *DEV2* are a counter and a d flip-flop. Describe the purpose of these devices in terms of how they are connected and what they are used for in the circuit.

The counter produces the addresses 0 – 7, incrementing by one on each CK cycle to copy a word to the main memory. When the address reaches 8, the flip-flop gets set to 1, which disables the counter because the copy of the block of 8 words is complete at that point.