EECS 370

Final Review

Announcements

- **Final exam** Tuesday 12/12 @ 10:30 – 12:30 am

- As with the midterm, you can have one 8.5 by 11 inch piece of paper with notes.
  - We will assume you know (or have written on your page) LC2K instructions (though not LC2K encodings).
    - So you should know what “lw 0 3 Bob” does.
- Room assignments on Piazza soon

Important topics

- Covers **all** course material with **emphasis pipelining and afterwards.**
  - Pipelining, data, and control hazards
  - Caches
  - Virtual memory
  - Performance

- Material presented in lectures, homeworks, labs, projects

Exam Strategy

- Pace yourself

- **Do not spend all your time on a single question**
  - If you don't understand a question, move ahead & come back to it later
- Answer as many questions as possible
  - Give yourself a few minutes at the end to check your work and/or attempt to solve easy questions that you have not attacked yet

Preparing for the exam

- Review the lectures
- Know how to solve the homework problems
- Practice exams w/solutions are posted to course website
  - **Complete without looking at the solutions!**

Best way to review for the exam

Practice,

Practice,

Practice

Plenty of choices: Old Exams, Class Problems, Midterm Review Problems

Selected Practice Questions

Just because it is not covered in this review it does not mean that it is not important

This review focuses on the newest material (because you've had less practice with it)

First question is warmup ☺

Problem 1: Multiple Choice (first 12…)

1. In ARM, the first 4 parameters to a function are stored in registers and the remaining parameters are stored in the ***stack / heap / data / virtual*** section of memory.

2. ***Moore's / Mealy's / Dennard's / Immerman's*** Law states that the number of transistors on a single chip will double every 2 years.

3. All instructions in a ***CISC / RISC / multi-core*** architecture (usually) have the same length.

Problem 1: Multiple Choice (first 12…)

4. The ***compiler / assembler / linker / loader*** is used to combine multiple object files into an executable program.

5. The ***compiler / assembler / linker / loader*** is used every time an executable program is run.

Problem 1: Multiple Choice (first 12…)

7. The output of ***combinational / sequential*** circuits depends exclusively on the current input.

8. Functions that don't call any functions (leaf functions) need not save and restore ***caller save / callee save / odd numbered*** registers.

9. If a cache includes dirty bits, then it must be using a ***write-back / write-through / allocate-on-write / not allocate-on-write*** policy.

Problem 1: Multiple Choice (first 12…)

10. The ***single-cycle / multi-cycle / pipelined*** datapath has the lowest CPI among the three types of datapath discussed in class.

11. A 1-way set associative cache is the same as a ***fully associative / direct mapped / virtually addressed / physically addressed*** cache.

12. Direct-mapped caches reduce the cost of finding the correct data by minimizing the number of **block offset / line index / tag / LRU** comparisons.

Problem 2 – Virtual Memory

Assume the following:

Virtual address size : 128 bytes;

Physical memory size : 8 pages;

Page replacement policy : LRU

Page size : 16 bytes

One-level page table of size : 16 bytes

Notes:

- On a page fault, the page table is updated before allocating a physical page.
- If more than one free page is available, the smallest physical page number is chosen.
- Physical page #0 is reserved for the operating system (OS). It cannot be replaced.
- If no mapping is found for a given virtual page assume the data is brought in from the disk.

Problem 2 – Virtual Memory (cont.)

a. The initial state of physical memory is shown on the left. Complete the page table for Process ID 11 (PID: 11) on the right.

Physical Page # (PPN)	Memory Contents
0x0	Reserved for OS
0x1	Page Table of PID 11
0x2	PID 11: VPN 0
0x3	
0x4	PID 11: VPN 4
0x5	PID 11: VPN 7
0x6	
0x7	

Page Table of PID:11

Virtual Page # (VPN)	Valid	Physical Page # (PPN)
0x0		
0x1		
0x2		
0x3		
0x4		
0x5		
0x6		
0x7		

Problem 2 – Virtual Memory (solution)

a. The initial state of physical memory is shown on the left. Complete the page table for Process ID 11 (PID: 11) on the right.

Physical Page # (PPN)	Memory Contents
0x0	Reserved for OS
0x1	Page Table of PID 11
0x2	PID 11: VPN 0
0x3	
0x4	PID 11: VPN 4
0x5	PID 11: VPN 7
0x6	
0x7	

Page Table of PID:11

Virtual Page # (VPN)	Valid	Physical Page # (PPN)
0x0	1	0x2
0x1	0	
0x2	0	
0x3	0	
0x4	1	0x4
0x5	0	
0x6	0	
0x7	1	0x5

Problem 2 – Virtual Memory (cont.)

b. Complete the following table for the given sequence of virtual address requests. Assume that the initial physical memory and page table state is shown in part a, above. **Process 7 begins at Time 1. Process 11 begins before Time 0, and ends after Time 5.*

Physical Page # (PPN)	Memory Contents
0x0	Reserved for OS
0x1	Page Table of PID 11
0x2	PID 11: VPN 0
0x3	Page Table of PID 7
0x4	PID 11: VPN 4
0x5	PID 11: VPN 7
0x6	
0x7	

Time	PID	Virt. Addr (VA)	Virtual Page # (VPN)	Physical Page # (PPN)	Page Fault? (Y/N)	Physical Address (PA)
0	11	0x0A				
1*	7	0x0A				
2	7	0x0B				
3	11	0x21				
4	7	0x74				
5*	7	0x35				
6	7	0x20				

Problem 2 – Virtual Memory (cont.)

b. Complete the following table for the given sequence of virtual address requests. Assume that the initial physical memory and page table state is shown in part a, above. **Process 7 begins at Time 1. Process 11 begins before Time 0, and ends after Time 5.*

Physical Page # (PPN)	Memory Contents
0x0	Reserved for OS
0x1	Page Table of PID 11
0x2	PID 11: VPN 0
0x3	Page Table of PID 7
0x4	PID 11: VPN 4
0x5	PID 11: VPN 7
0x6	
0x7	

Time	PID	Virt. Addr (VA)	Virtual Page # (VPN)	Physical Page # (PPN)	Page Fault? (Y/N)	Physical Address (PA)
0	11	0x0A	0x0	0x2	N	0x2A
1*	7	0x0A	0x0	0x6	Y	0x6A
2	7	0x0B	0x0	0x6	N	0x6B
3	11	0x21	0x2	0x7	Y	0x71
4	7	0x74	0x7	0x4	Y	0x44
5*	7	0x35	0x3	0x5	Y	0x55
6	7	0x20	0x2	0x1	Y	0x10

Problem 3: Branch Prediction (F16)

Consider the following C code and corresponding LC2K assembly. Assume it is executed on a pipelined data-path with branch speculation discussed in class.

```
int i, j, x = 0;
for (i=0; i != 30; i++) {
      for (j=0; j != 2; j++) {
            x++;
      }
}
```

```
      lw   0  1  cnt1
      lw   0  2  cnt2
      lw   0  3  one
outer noop
      beq  0  1  exit        //B1
      noop
      lw   0  2  cnt2
inner beq  0  2  done        //B2
      add  4  3  4
      add  2  3  2
      beq  0  0  inner       //B3
done  add  1  3  1
      beq  0  0  outer       //B4
exit halt
cnt1 .fill 30
cnt2 .fill 2
one .fill -1
```

a) How many branches are executed?
b) Accuracy for predict not-taken?
c) Accuracy for local 1-bit predictor? (Each branch has its own predictor)

a) How many branch instructions are executed for a single run of the program?

Problem 3: Branch Prediction (F16)

Consider the following C code and corresponding LC2K assembly. Assume it is executed on a pipelined data-path with branch speculation discussed in class.

```
int i, j, x = 0;
for (i=0; i != 30; i++) {
      for (j=0; j != 2; j++) {
            x++;
      }
}
```

```
      lw   0  1  cnt1
      lw   0  2  cnt2
      lw   0  3  one
outer noop
      beq  0  1  exit        //B1
      noop
      lw   0  2  cnt2
inner beq  0  2  done        //B2
      add  4  3  4
      add  2  3  2
      beq  0  0  inner       //B3
done  add  1  3  1
      beq  0  0  outer       //B4
exit halt
cnt1 .fill 30
cnt2 .fill 2
one .fill -1
```

a) How many branch instructions are executed for a single run of the program?

B1: 31 B2: 90 B3: 60 B4: 30

Problem 3: Branch Prediction (F16)

Consider the following C code and corresponding LC2K assembly. Assume it is executed on a pipelined data-path with branch speculation discussed in class.

```
int i, j, x = 0;
for (i=0; i != 30; i++) {
     for (j=0; j != 2; j++) {
          x++;
     }
}
```

```
      lw   0  1  cnt1
      lw   0  2  cnt2
      lw   0  3  one
outer noop
      beq  0  1  exit       //B1
      noop
      lw   0  2  cnt2
inner beq  0  2  done       //B2
      add  4  3  4
      add  2  3  2
      beq  0  0  inner      //B3
done  add  1  3  1
      beq  0  0  outer      //B4
exit halt
cnt1 .fill 30
cnt2 .fill 2
one .fill -1
```

b) How many branches are predicted correctly if we predict Always-Not-Taken?

Problem 3: Branch Prediction (F16)

Consider the following C code and corresponding LC2K assembly. Assume it is executed on a pipelined data-path with branch speculation discussed in class.

```
int i, j, x = 0;
for (i=0; i != 30; i++) {
     for (j=0; j != 2; j++) {
          x++;
     }
}
```

```
      lw   0  1  cnt1
      lw   0  2  cnt2
      lw   0  3  one
outer noop
      beq  0  1  exit       //B1
      noop
      lw   0  2  cnt2
inner beq  0  2  done       //B2
      add  4  3  4
      add  2  3  2
      beq  0  0  inner      //B3
done  add  1  3  1
      beq  0  0  outer      //B4
exit halt
cnt1 .fill 30
cnt2 .fill 2
one .fill -1
```

b) How many branches are predicted correctly if we predict Always-Not-Taken?

NT executions: B1: 30 B2: 60 B3: 0 B4: 0

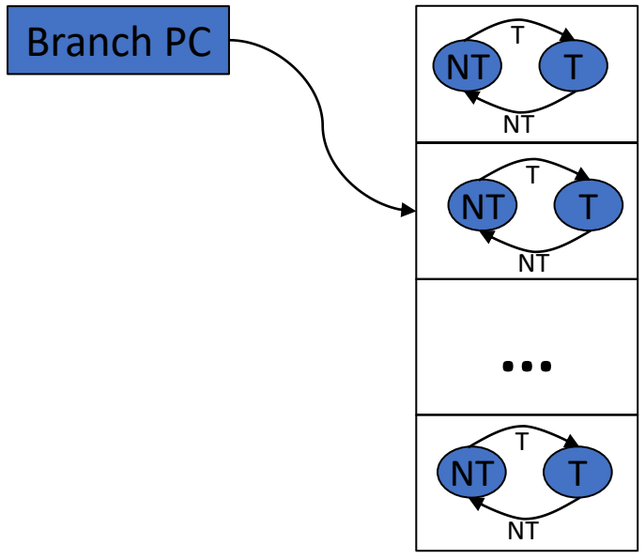
Total executions: B1: 31 B2: 90 B3: 60 B4: 30

Problem 3: Branch Prediction (F16)

c) How many branches are predicted correctly if we use local last time 1-bit predictor?

Local 1-bit Predictor: In this prediction scheme, the PC of a branch instruction is used to index into a table. Each table entry for a branch is a 1-bit predictor, which predicts the same outcome as last time for that branch. Assume that the entries are initialized to Not Taken.

Local Last Time Predictor Table




```
int i, j, x = 0;
for (i=0; i != 30; i++) {
      for (j=0; j != 2; j++) {
            x++;
      }
}
```

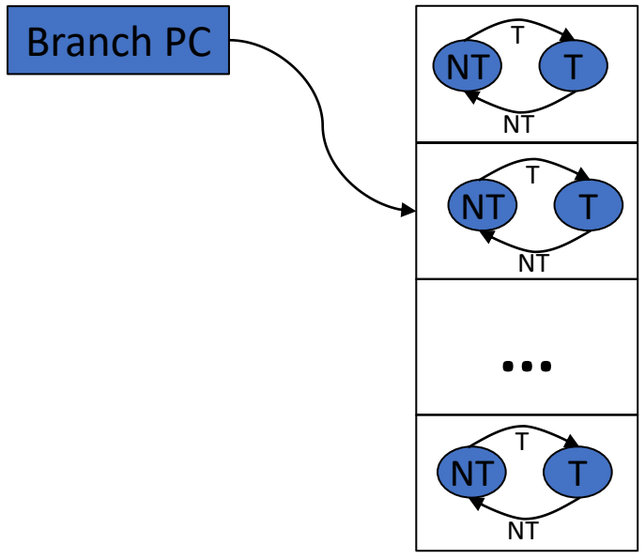
```
      lw 0 1 cnt1
      lw 0 2 cnt2
      lw 0 3 one
outer noop
      beq 0 1 exit        B1
      noop
      lw 0 2 cnt2
inner beq 0 2 done        B2
      add 4 3 4
      add 2 3 2
      beq 0 0 inner       B3
done  add 1 3 1
      beq 0 0 outer       B4
exit halt
cnt1 .fill 30
cnt2 .fill 2
one .fill -1
```

Problem 3: Branch Prediction (F16)

c) How many branches are predicted correctly if we use local last time 1-bit predictor?

Local 1-bit Predictor: In this prediction scheme, the PC of a branch instruction is used to index into a table. Each table entry for a branch is a 1-bit predictor, which predicts the same outcome as last time for that branch. Assume that the entries are initialized to Not Taken.

Local Last Time Predictor Table



B1: 30 B2: 31 B3: 59 B4: 29

```
int i, j, x = 0;
for (i=0; i != 30; i++) {
      for (j=0; j != 2; j++) {
            x++;
      }
}
```

```
      lw 0 1 cnt1
      lw 0 2 cnt2
      lw 0 3 one
outer noop
      beq 0 1 exit        B1
      noop
      lw 0 2 cnt2
inner beq 0 2 done        B2
      add 4 3 4
      add 2 3 2
      beq 0 0 inner       B3
done  add 1 3 1
      beq 0 0 outer       B4
exit halt
cnt1 .fill 30
cnt2 .fill 2
one .fill -1
```

Problem 4: Multi-level Page Tables

You need to design a 3-level hierarchical page table. Physical memory is with 8KB (byte-addressable); virtual memory addresses are 17-bits long. A page is 32 bytes & each page table entry is 2bytes. The 1st level table & each of the 2nd level & 3rd level page tables require precisely one page of storage.

a. Determine the bit positions for the following fields in a virtual address and a physical address.

Virtual Address			
1st level offset	2nd level offset	3rd level offset	Page offset
Bits:	Bits:	Bits:	Bits:

Physical Address	
Physical Page Number	Page offset
Bits:	Bits:

Problem 4: Multi-level Page Tables (sol.)

You need to design a 3-level hierarchical page table. Physical memory is with 8KB (byte-addressable); virtual memory addresses are 17-bits long. A page is 32 bytes & each page table entry is 2bytes. The 1st level table & each of the 2nd level & 3rd level page tables require precisely one page of storage.

a. Determine the bit positions for the following fields in a virtual address and a physical address. Page offset field has been completed as an example.

Virtual Address			
1st level offset	2nd level offset	3rd level offset	Page offset
Bits:16-13	Bits:12-9	Bits:8-5	Bits:4-0

Physical Address	
Physical Page Number	Page offset
Bits:12-5	Bits:4-0

Problem 4: Multi-level Page Tables

b. Assume that at the beginning of a program's execution only the 1st level table is in main memory. For the program shown below, calculate the total size of all page tables (including the 1st level, 2nd level and 3rd level page tables) in main memory after executing the `for` loop. Assume the array `BigArray` is allocated at the virtual address 0x0 and '`char`' = 1 byte. Show your work.

```
char BigArray[2048*32];
for (i = 0; i < 2048; i += 16){
      sum += BigArray[i*32];
}
```

Total 2nd Level Page Tables?

Total 3rd Level Page Tables?

Total Size in memory?

Problem 4: Multi-level Page Tables

b.

```
char BigArray[2048*32];
for (i = 0; i < 2048; i += 16){
      sum += BigArray[i*32];
}
```

Problem 4: Multi-level Page Tables (sol.)

b.

```
char BigArray[2048*32];
for (i = 0; i < 2048; i += 16){
      sum += BigArray[i*32];
}
```

Total 2nd Level Page Tables? 8

Total 3rd Level Page Tables? 128

Total Size in memory? (1+8+128)*32Bytes = 4384Bytes

Now you are a Computer Architect (yay!)

- One architecture, many microarchitectures
- If you like this stuff, take EECS 470, 482
- Study for the final & have a fun summer!

"Leading a student to learning's treasures
Gives a teacher untold pleasures!"

What's next for You?!

- What classes should you look at if your interested in this kind of stuff?

More Classes?

- EECS 470 – Computer Architecture
- Picks up where we leave off in 370
- Discuss more sophisticated enhancements to processor design
  - Out-of-order execution
  - Multi-core / multi-threaded processors
- Major Design Experience / Capstone
  - Work in teams to design an actual processor (not simulator) over the course of the semester
  - **LOT'S** of work
- Requires EECS 270
  - For Verilog (a Hardware Description Language)

More Classes?

- EECS 373 – Embedded Systems
  - Learn about parts of computer systems that aren't in the processor or cache
    - Input/output
    - Timers
    - Bus interfaces
  - Followed up by 473 (MDE / Capstone)

- EECS 471 – Applied Parallel Programming with GPUs
  - How are graphics processing units different than normal processors?
  - How do we take advantage of their raw power when writing software?

More Classes?

- EECS 427 – VLSI Design
  - Design a processor at the circuit level
  - Actually layout all the transistors
  - Less architecture – more circuit design

- EECS 570 (Parallel Computer Architecture) + 573 (Microarchitecture)
  - Learn about more current research into architecture
  - Requires 470

More Classes?

- EECS 483 (Front-end compilers) + 583 (Back-end compilers)
  - 583 is more relevant to architecture
  - How to design compilers to write efficient assembly?
  - Requires knowledge of hardware optimizations
- EECS 482 – Operating Systems
  - How do moderns systems support multiple threads running simultaneously?
  - How does OS manage virtual memory?
  - How do file systems work?
- EECS 388 – Computer Security
  - How do things like caches "leak" information about a program's data?
  - How can call stacks be tricked into executing arbitrary code?

More Classes?

- EECS 498-001 – Quantum Computing for the Computer Scientist
  - Winter 24
  - How can we redesign the computing stack to take advantage of the bizarre rules of quantum mechanics?
  - Rather than having bits be 0 or 1, have quantum bits be in **superpositions** of 0 and 1
    - Perform massive number of computations simultaneously… sort of
  - Requires 370 and 281

Other questions?!