# EE 457 Final Review
## Redekopp

1. **Virtual Memory**: Given a virtual memory system with 24-bit virtual addresses, 1 KB pages, and 20-bit physical addresses.
   a. How many address bits will be used for the page offset field? _1KB => 10-bits_

   b. How many bits will each Physical Page Frame Number require? __20-10=10-bits

   c. Given an 8-way set associative TLB with 256 entries, perform the virtual address breakdown (i.e. how many address bits will be used for the set field and for the tag field of the address mapping. Show any work.

      256 entries / 8-ways per set
      = 32 sets => 5-bits

      Reference Layout of TLB Address Mapping:

      | Tag | Set | Page offset |
      |---|---|---|

      Number of set bits: __5 set bits____

      Number of tag bits: __24-5-10 = 9 tag bits_

   d. Given a single level page table, how much memory would be required to hold the table assuming each entry in the table requires **4 bytes** (this includes the page frame, valid, dirty and other bits). [Hint: Size = # of entries * bytes per entry]

      Number of entries => 24-10 = 14-bits for VPN
      => 14-bits => $2^{14}$ entries

      Size = $2^{14}$ entries * 4 bytes each = 64 KB

   e. Given a two level page table where the 1st level has 32 entries and the 2nd level contains the rest of the needed entries, find a sequence of 3 memory accesses that would require accessing 2nd level page tables 0, 1, and 31 to be allocated.

      24 VPN bits divided into:

      32 entries of 1st level => 5 bits

      14 – 5 = 9-bits for 2nd level

      Access to 2nd level page table =>

      0000 0000 0000 0000 0000 0000

      0000 1000 0000 0000 0000 0000

      1111 1000 0000 0000 0000 0000

2. **Cache.** Examine the following sequence of memory accesses.

| | | |
|---|---|---|
| 1. Read 0x0a0<br>2. Write 0x0b4<br>3. Read 0x124<br>4. Write 0x170<br>5. Read 0x33c<br>6. Read 0x128<br>7. Write 0x4ac<br>8. Read 0x33c<br>9. Read 0x4b0 | | Assumptions<br>• 12-bit byte addresses<br>• Word accesses only<br>• Cache Size and Block size = **8 blocks** of **8 words** each<br>• No-Load-Through<br>• Write-back<br>• 2 way Set-Associative Cache |

a. Perform the address breakdown for the given cache configuration

| Block Range | **Tag = 5** | **Set = 2** | **Word = 3** | **Unused = 2** |
|---|---|---|---|---|
| 0a0 – 0bc | 00001 | 01 | … | |
| 120 – 13c | 00010 | 01 | … | |
| 160-17c | 00010 | 11 | … | |
| 320 – 33c | 00110 | 01 | … | |
| 4a0-4bc | 01001 | 01 | … | |

b. Now list the block operations the 2-way set-associative cache will perform for each access. Possible block operations are: *Fetch Block XX-YY, Evict Block XX-YY w/o writeback, Evict Block XX-YY w/ Writeback, Final Writeback of Block XX-YY, where XX-YY is the block address range.* Hits do not require any block operation. Hint 1: Each access may required 0-2 block ops.
Hint 2: It will help to keep track of which blocks are in the cache.

| | |
|---|---|
| 1. Read 0x0a0<br>Fetch block 0x0a0-0x0bf => Set 1 | 6. Read 0x128<br>Hit |
| 2. Write 0x0b4<br>Hit | 7. Write 0x4ac<br>Evict block 0x320-0x33f w/o WB<br>Fetch block 0x4a0-4bf => Set 1 |
| 3. Read 0x124<br>Fetch block 0x120-0x13f => Set 1 | 8. Read 0x33c<br>Evict block 0x120-0x13f w/o WB<br>Fetch block 0x320-33f => Set 1 |
| 4. Write 0x170<br>Fetch block 0x160-0x17f => Set 3 | 9. Read 0x4b0 |
| 5. Read 0x33c<br>Evict 0x0a0-0x0bf w/ WB<br>Fetch block 0x320-0x33f = Set 1 | Final Writebacks:<br>Final WB of 0x4a0-0x4bf<br>Final WB of 160-17f |

3.) (12 pts.) Given the code below, perform explicit register renaming to solve all WAW, WAR hazards present in the original code. When performing register renaming, use register numbers $10, $11, $12… in that order so that everyone's answer will hopefully be more uniform.

| | |
|---|---|
| `lw $5,0($2)`<br>`add $6,$4,$5`<br>`sub $7,$7,$6`<br>`lw $4,0($3)`<br>`sub $6,$4,$2`<br>`add $3,$7,$2` | `lw $5,0($2)`<br>`add $6,$4,$5`<br>`sub $7,$7,$6`<br>`lw $10,0($3)`<br>`sub $11,$10,$2`<br>`add $12,$7,$2` |

4.) (5 pts.) Given the code below, (same as in Question 1, assume the first 'lw' instruction stalls due to a cache miss. Assuming an out-of-order, dynamically scheduled processor (that performs automatic register renaming), which instructions would be allowed to execute and which instructions would need to stall due to the 'lw'.

| Code | Circle the correct answer |
|---|---|
| `lw $5,0($2)` | CACHE MISS |
| `add $6,$4,$5` | **Stall** / Execute |
| `sub $7,$7,$6` | **Stall** / Execute |
| `lw $4,0($3)` | Stall / **Execute** |
| `sub $6,$4,$2` | Stall / **Execute** |
| `add $3,$7,$2` | **Stall** / Execute |

5.) (22 pts.) Given the code below, (same as in Question 1, *assuming all functional units are currently stalled* (none of the instructions below can execute) show the state of the register status table after each instruction issues. Then show what source operands will be in each reservation station (operand value or RS name of producer) for each instruction. Use reservation station names (A1, A2, S1, S2, and L1, L2) in both the reservation stations and the register status table ('-' in the table means blank) . Assume the following initial values for each register: R[2] = 0x02, R[3] = 0x03, R[4] = 0x04, R[5] = 0x05, R[6] = 0x06, R[7] = 0x07.

| Code | Register Status Table (After Exec. Of Each Instruction) | | | | | |
|---|---|---|---|---|---|---|
| | $2 | $3 | $4 | $5 | $6 | $7 |
| lw $5,6($2) | - | - | - | L1 | - | - |
| add $6,$4,$5 | - | - | - | L1 | A1 | - |
| sub $7,$7,$6 | - | - | - | L1 | A1 | S1 |
| lw $4,5($3) | - | - | L2 | L1 | A1 | S1 |
| sub $6,$4,$2 | | | L2 | L1 | S2 | S1 |
| add $3,$7,$6 | | A2 | L2 | L1 | S2 | S1 |