

# CS61C Summer 2014 Final Review

Andrew Luo

## Agenda

- CALL
- Virtual Memory
- Data Level Parallelism
- Instruction Level Parallelism
- Break
- Final Review Part 2 (David)

## CALL

- CALL:
  - Compiler
  - Assembler
  - Linker
  - Loader

## CALL

- Compiler
  - Takes high-level code (such as C) and creates assembly code
- Assembler
  - Takes assembly code and creates intermediate object files
- Linker
  - Links intermediate object files into executable/binary
- Loader
  - Runs the executable/binary on the machine; prepares the memory structure

## Compiler

- Project 1
- Takes a high level language (such as C or C++) and compiles it into a lower-level, machine-specific language (such as x86 ASM or MIPS ASM)
- Different than an interpreter!

## Compilation vs Interpretation

- C is a compiled language, whereas C#, Java, and Python are interpreted (Java is a little different actually but is interpreted in the end)
- Technically an implementation detail, as languages are just semantics; theoretically it would be possible to interpret C and compile C#/Java/Python, but this is rare/odd in practice.

### What are some advantages/disadvantages of compilation and interpretation?

- First, you tell me!

### What are some advantages/disadvantages of compilation and interpretation?

- Compilation is faster
- Generally interpreted languages are higher-level and easier to use
- Interpretation is simpler/easier
- Interpretation generates smaller code
- Interpretation is more machine independent

### Assembler

- Assembles assembly language code into object files
- Fairly basic compared to the compiler
- Usually a simple 1:1 translation from assembly code to binary

### Assembler Directives

- Who knows these assembler directives?
- .text
- .data
- .globl sym
- .ascii z
- .word

### Assembler Directives

- Who knows these assembler directives?
- .text: text segment, code
- .data: data segment, binary data
- .globl sym: global symbols that can be exported to other files
- .ascii z: ASCII strings
- .word: 32-bit words

### Assembler: Branches and Jumps

- How are these handled by the assembler?

## Assembler: Branches and Jumps

- First run through the program and change and psuedoinstructions to the corresponding real instructions.
  - Why do this?

## Assembler: Branches and Jumps

- First run through the program and change and psuedoinstructions to the corresponding real instructions.
  - Some psuedoinstructions actually become 2 or more instructions so will change the absolute and/or relative addresses of branch and/or jump targets
- Next, convert all the labels to addresses and replace them
  - Branches are PC-relative
  - Jumps are absolute addressed

## Linker

- Link different object files together to create an executable
- Must resolve address conflicts in different files
  - Relocate code -> change addresses

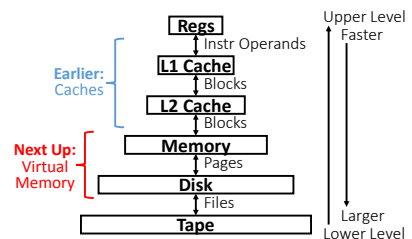
## Loader

- Handled by the operating system (and by the C Runtime)
- Prepares memory resources, such as initializing the stack pointer, allocating the necessary pages for heap, stack, static, and text segments.

## Agenda

- CALL
- **Virtual Memory**
- Data Level Parallelism
- Instruction Level Parallelism
- Break
- Final Review Part 2 (David)

## Memory Hierarchy



8/9/2014

Summer 2014 - Lecture 25

18

### Memory Hierarchy Requirements

- Principle of Locality
  - Allows caches to offer (close to) speed of cache memory with size of DRAM memory
  - Can we use this at the next level to give speed of DRAM memory with size of Disk memory?
- What other things do we need from our memory system?

8/31/2014 Summer 2014 - Lecture 23 19

### Memory Hierarchy Requirements

- Allow multiple processes to simultaneously occupy memory and provide **protection**
  - Don't let programs read from or write to each other's memories
- Give each program the illusion that it has its own **private address space**
  - Suppose a program has base address 0x00400000, then different processes each think their code resides at the same address
  - Each program must have a different view of memory

8/31/2014 Summer 2014 - Lecture 23 20

### Virtual Memory

- Next level in the memory hierarchy
  - Provides illusion of very large main memory
  - Working set of "pages" residing in main memory (subset of all pages residing on disk)
- Main goal:** Avoid reaching all the way back to disk as much as possible
- Additional goals:**
  - Let OS share memory among many programs and protect them from each other
  - Each process thinks it has all the memory to itself

8/31/2014 Summer 2014 - Lecture 23 21

### Virtual to Physical Address Translation

- Each program operates in its own virtual address space and thinks it's the only program running
- Each is protected from the other
- OS can decide where each goes in memory
- Hardware gives virtual → physical mapping

8/31/2014 Summer 2014 - Lecture 23 22

### Mapping VM to PM

- Divide into equal sized chunks (usually 4-8 KiB)
- Any chunk of Virtual Memory can be assigned to any chunk of Physical Memory ("page")

8/31/2014 Summer 2014 - Lecture 23 23

### Address Mapping

- Pages are aligned in memory
  - Border address of each page has same lowest bits
  - Page size (P bytes) is same in VM and PM, so denote lowest  $PO = \log_2(P)$  bits as **page offset**
- Use remaining upper address bits in mapping
  - Tells you which page you want (similar to Tag)

8/31/2014 Summer 2014 - Lecture 23 24

### Page Table Entry Format

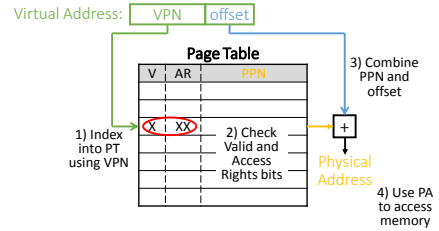
- Contains either PPN or indication not in main memory
- **Valid** = Valid page table entry
  - 1 → virtual page is in physical memory
  - 0 → OS needs to fetch page from disk
- **Access Rights** checked on every access to see if allowed (provides protection)
  - *Read Only*: Can read, but not write page
  - *Read/Write*: Read or write data on page
  - *Executable*: Can fetch instructions from page

8/31/2014

Summer 2014 - Lecture 23

25

### Page Table Layout



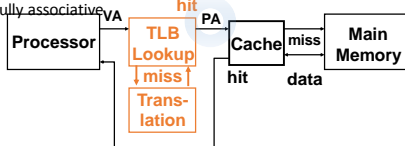
8/31/2014

Summer 2014 - Lecture 23

26

### Translation Look-Aside Buffers (TLBs)

- TLBs usually small, typically 128 - 256 entries
- Like any other cache, the TLB can be direct mapped, set associative, or fully associative



On TLB miss, get page table entry from main memory

### Context Switching and VM

- What happens in the case of a context switch?

### Context Switching and VM

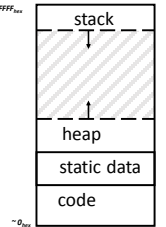
- We need to flush the TLB
- Do we need to flush the cache?

### Context Switching and VM

- We need to flush the TLB as they are in virtual addresses
  - In reality we can use context tagging
- Do we need to flush the cache?
  - No, if using physical addresses
  - Yes, if using virtual addresses

### Why would a process need to "grow"?

- A program's *address space* ~### ###... contains 4 regions:
  - stack**: local variables, grows downward
  - heap**: space requested for pointers via `malloc()`; resizes dynamically, grows upward
  - static data**: variables declared outside main, does not grow or shrink
  - code**: loaded when program starts, does not change



What is the grey area between the stack and the heap?

### Practice Problem

- For a 32-bit processor with 256 KiB pages and 512 MiB of main memory:
  - How many entries in each process' page table?
  - How many PPN bits do you need?
  - How wide is the page table base register?
  - How wide is each page table entry? (assume 4 permission bits)

### Practice Problem

- For a 32-bit processor with 256 KiB pages and 512 MiB of main memory:
  - How many entries in each process' page table?
    - 256 KiB -> 18 offset bits, 32 - 18 = 14 VPN bits, 2<sup>14</sup> entries
  - How PPN bits?
    - 512 MiB / 256 KiB = 2<sup>29</sup> / 2<sup>18</sup> = 2<sup>11</sup> pages, 11 PPN bits
  - How wide is the page table base register?
    - log(512 MiB) = 29
  - How wide is each page table entry? (assume 4 permission bits)
    - 4 (permission) + 11 (PPN) + 1 (valid) + 1 (dirty) = 17

### Agenda

- CALL
- Virtual Memory
- Data Level Parallelism**
- Instruction Level Parallelism
- Break
- Final Review Part 2 (David)

### SIMD

- Who knows what SIMD is?

### SIMD

- Who knows what SIMD is?
  - Single Instruction Multiple Data

## SIMD

- MIMD, MISD, SISD?
- Examples of each?

## SSE Problem

```
float* add(float* a, float* b, size_t n)
{

}
```

## SSE Problem

```
float* add(float* a, float* b, size_t n)
{
    float* result = malloc(sizeof(float) * n);
}
```

## SSE Problem

```
float* add(float* a, float* b, size_t n)
{
    float* result = malloc(sizeof(float) * n);
    for (size_t i = 0; i < n - 3; i += 4)
    {
        _mm_storeu_ps(result, _mm_add_ps(_mm_loadu_ps(a + i), _mm_loadu_ps(b + i)));
    }
}
```

## SSE Problem

```
float* add(float* a, float* b, size_t n)
{
    float* result = malloc(sizeof(float) * n);
    size_t i = 0;
    for (; i < n - 3; i += 4)
    {
        _mm_storeu_ps(result, _mm_add_ps(_mm_loadu_ps(a + i), _mm_loadu_ps(b + i)));
    }
    for (; i < n; i++)
    {
        result[i] = a[i] + b[i];
    }
    return result;
}
```

## Agenda

- CALL
- Virtual Memory
- Data Level Parallelism
- **Instruction Level Parallelism**
- Break
- Final Review Part 2 (David)

## Multiple Issue

- Modern processors can issue and execute multiple instructions per clock cycle
- $CPI < 1$  (*superscalar*), so can use *Instructions Per Cycle* (IPC) instead
- e.g. 4 GHz 4-way multiple-issue can execute 16 billion IPS with peak  $CPI = 0.25$  and peak  $IPC = 4$ 
  - But dependencies and structural hazards reduce this in practice

8/31/2014

Summer 2014 - Lecture 23

43

## Multiple Issue

- Static multiple issue
  - Compiler reorders independent/commutative instructions to be issued together
  - Compiler detects and avoids hazards
- Dynamic multiple issue
  - CPU examines pipeline and chooses instructions to reorder/issue
  - CPU can resolve hazards at runtime

8/31/2014

Summer 2014 - Lecture 23

44

## Agenda

- CALL
- Virtual Memory
- Data Level Parallelism
- Instruction Level Parallelism
- Break
- Final Review Part 2 (David)