## Lecture 26 (Fri 12/05/2008)

- HW #4 (optional) - Due NOW
- Lab #4 Hardware - Due Fri Dec 5 at 5pm

- **Final Exam, Monday Dec. 8th, 8:30-10:20 in EEB 037 (here, our regular lecture room)** Bring highlighter.

- **Today**: Wrap Up!

## Final Exam

- The final is comprehensive, but most coverage is on material since midterm (I will provide green card again, could have MIPS programming)

  Microprogramming
  Pipelining
  - Overall operation
  - Hazards

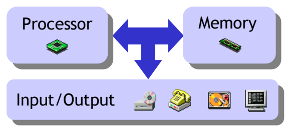
  Caching
  - Overall operation and design space

  Performance
  VM/Paging
  Interrupts/Exceptions
  I/O
  Parallelism (Extra Credit)

- Style is similar to MT: some shorter questions, some longer ones
- Reading is on the lectures page. Study the slides, review the reading, exercises in lecture, HW4.

## Instant replay

- This quarter was split into roughly four parts.
  - 1st we covered instruction set architectures—the connection between software and hardware.
  - 2nd we discussed processor design. We focused on pipelining, which is one of the most important ways of improving processor performance.
  - 3rd we focused on large and fast memory systems (via caching), virtual memory, and I/O.
  - Finally, we discussed performance tuning (HW3), and exploiting data parallelism via SIMD and Multi-Core processors.
- We also introduced many performance metrics to estimate the actual benefits of all of these fancy designs.



## Some recurring themes

- There were several recurring themes throughout the quarter:
  - Instruction set and processor designs are intimately related.
  - Parallel processing can often make systems faster.
  - Performance and Amdahl’s Law quantifies performance limitations.
  - Hierarchical designs combine different parts of a system.
  - Hardware and software depend on each other.

## Instruction sets and processor designs

- The MIPS instruction set was designed for pipelining.
  - All instructions are the same length, to make instruction fetch and jump and branch address calculations simpler.
  - Opcode and operand fields appear in the same place in each of the three instruction formats, making instruction decoding easier.
  - Only relatively simple arithmetic and data transfer instructions are supported.
- These decisions have multiple advantages.
  - They lead to shorter pipeline stages and higher clock rates.
  - They result in simpler hardware, leaving room for other performance enhancements like forwarding, branch prediction, and on-die caches.

5

## x86 Selected History

| Processor | Intro Year | Intro Clock | Transistors | Features |
|---|---|---|---|---|
| 8086 | 1978 | 8 MHz | 29 K | 16-bit regs., segments |
| 286 | 1982 | 12.5 MHz | 134 K | Protected mode |
| 386 | 1985 | 20 MHz | 275 K | 32-bit regs., paging |
| 486 | 1989 | 25 MHz | 1.2 M | On-board FPU |
| Pentium | 1993 | 60 MHz | 3.1 M | MMX on late models |
| Pentium Pro | 1995 | 200 MHz | 5.5 M | P6 core, bigger caches |
| Pentium II | 1997 | 266 MHz | 7 M | P6 w/MMX |
| Pentium III | 1999 | 700 MHz | 28 M | SSE (Streaming SIMD) |
| Pentium 4 | 2000 | 1.5 GHz | 42 M | NetBurst core, SSE2 |
| Xeon | 2001 | 2.2 GHz | 55 M | Hyper-Threading |
| Pentium M | 2003 | 1.6 GHz | 77 M | Shorter pipelines vs P4 |

6

## Registers in x86

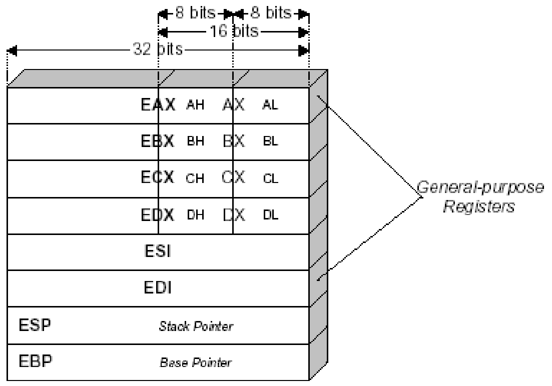


Figure 1. The x86 register set.

7

## Addressing Memory

<u>Specifying a memory address:</u>
up to 2 registers and 1 32-bit signed constant can be added together to compute a memory address. One register can be optionally pre-multiplied by 2,4,8.

mov eax, ebx
mov eax, [ebx]
mov [var], ebx
mov eax, [esi -4]
mov [esi+eax], cl
mov edx, [esi+4*ebx]

**Incorrect**: (why?)
mov eax, [ebx – ecx]
mov [eax+esi+edi], ebx
mov [4*eax+2*ebx], ecx

8

## Caller / Callee

```
push [var]
push 123
push eax

; call myFunc(eax,123,[var])
call _myFunc

add esp, 12
```

```
_myFunc PROC
push ebp
pov ebp, esp
sub esp, 4
push edi
push esi
mov eax, [ebp+8]
mov esi, [ebp+12]
mov edi, [ebp+16]
mov[ebp-4], edi
add[ebp-4], esi
add eax, [ebp-4]
pop esi
pop edi
mov esp, ebp
pop ebp
ret
ENDP _myFunc
```

## Parallel processing

- One way to improve performance is to do more processing at once.
- There were several examples of this in our CPU designs:
  - 
  - 
  - 
  - 
- Memory and I/O systems also provide many good examples:
  - 
  - 
  -

## Parallel processing

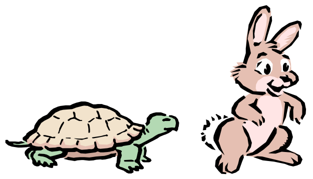
- One way to improve performance is to do more processing at once.
- There were several examples of this in our CPU designs.
  - Multiple functional units can be included in a datapath to let single instructions execute faster. For example, we can calculate a branch target while reading the register file.
  - Pipelining allows us to overlap the executions of several instructions.
  - SIMD performance operations on multiple data items simultaneously.
  - Multi-core processors enable thread-level parallel processing.
- Memory and I/O systems also provide many good examples.
  - A wider bus can transfer more data per clock cycle.
  - Memory can be split into banks that are accessed simultaneously. Similar ideas may be applied to hard disks, as with RAID systems.
  - A direct memory access (DMA) controller performs I/O operations while the CPU does compute-intensive tasks instead.

## Performance and Amdahl’s Law

- First Law of Performance: **Make the common case fast!**

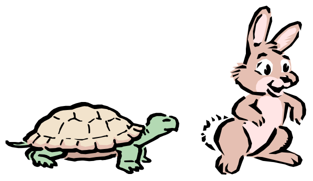
- But, performance is limited by the slowest component of the system.
- We’ve seen this in regard to cycle times in our CPU implementations:
  - 
  -

- Amdahl’s Law also holds true outside the processor itself:
  - 
  - 
  -



## Performance and Amdahl's Law

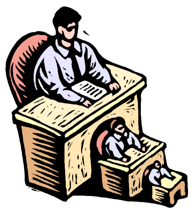
- First Law of Performance: **Make the common case fast!**
- But, performance is limited by the slowest component of the system.
- We've seen this in regard to cycle times in our CPU implementations.
  - Single-cycle clock times are limited by the slowest instruction.
  - Pipelined cycle times depend on the slowest individual stage.
- Amdahl's Law also holds true outside the processor itself.
  - Slow memory or bad cache designs can hamper overall performance.
  - I/O bound workloads depend on the I/O system's performance.



13

## Hierarchical designs

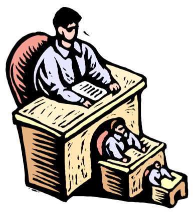
- Hierarchies separate fast and slow parts of a system, and minimize the interference between them.
  - 
  -



14

## Hierarchical designs

- Hierarchies separate fast and slow parts of a system, and minimize the interference between them.
  - Caches are fast memories which speed up access to frequently-used data and reduce traffic to slower main memory. (Registers are even faster…)
  - Buses can also be split into several levels, allowing higher-bandwidth devices like the CPU, memory and video card to communicate without affecting or being affected by slower peripherals.



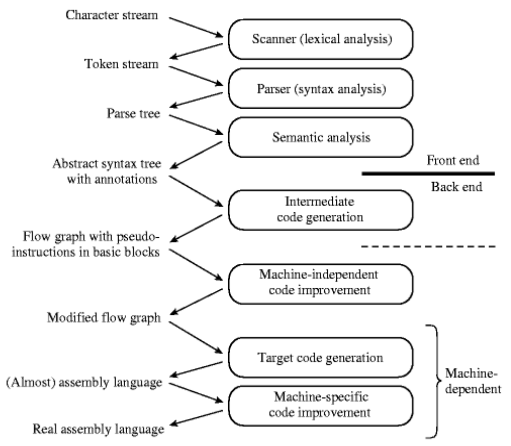
15

## Architecture and Software

- Computer architecture plays a vital role in many areas of software.
- Compilers are critical to achieving good performance.
  - They must take full advantage of a CPU's instruction set.
  - Optimizations can reduce stalls and flushes, or arrange code and data accesses for optimal use of system caches.
- Operating systems interact closely with hardware.
  - They should take advantage of CPU features like support for virtual memory and I/O capabilities for device drivers.
  - The OS handles exceptions and interrupts together with the CPU.

16

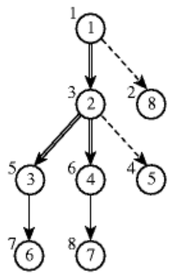
## Compiler Structure


Character stream
Scanner (lexical analysis)
Token stream
Parser (syntax analysis)
Parse tree
Semantic analysis
Front end
Back end
Abstract syntax tree with annotations
Intermediate code generation
Flow graph with pseudo-instructions in basic blocks
Machine-independent code improvement
Modified flow graph
Target code generation
(Almost) assembly language
Machine-specific code improvement
Real assembly language
Machine-dependent

## Instruction Scheduling

- **Goal**: Find a schedule of instructions that is optimal for a pipelined machine
- First organize instructions in a DAG (instructions, dependences).
  - Flow dependence
  - Anti-dependence
  - Output dependence
- Any topological sort of the graph is legal, but we want one that minimizes overall delay (NP-hard).
- So instead use heuristics to traverse and minimize as much as possible.

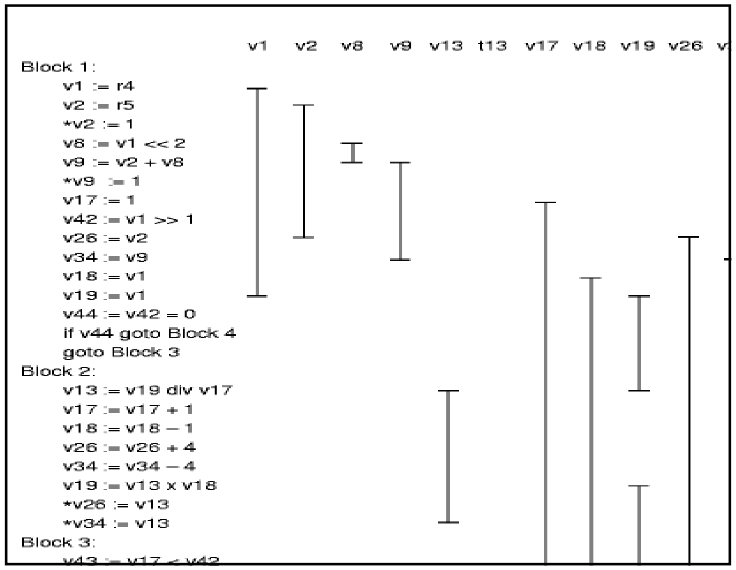
```
Block 2:                              Scheduled:
 1.  v19 := v13 x v18                 v19 := v13 x v18
     —                                v18 := v18 − 1
     —                                    —
     —                                    —
     —                                    —
 2.  v13 := v19 div v17               v13 := v19 div v17
     —                                v17 := v17 + 1
     —                                    —
     —                                    —
     —                                    —
 3.  *v26 := v13                      *v26 := v13
 4.  *v34 := v13                      *v34 := v13
 5.  v17 := v17 + 1                   v26 := v26 + 4
 6.  v26 := v26 + 4                   v34 := v34 − 4
 7.  v34 := v34 − 4
 8.  v18 := v18 − 1
     −− fall through to Block 3
                                      (same)
Block 3:
    v43 := v17 ≤ v42
    if v43 goto Block 2
    −− else fall through to Block 4
```



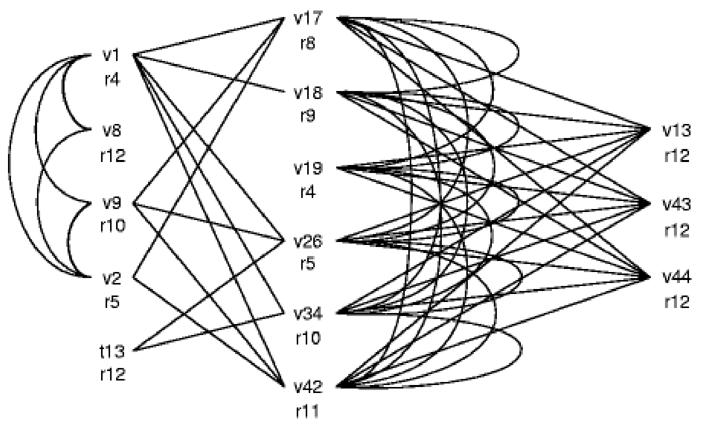
## Register Allocation

- Equivalent to graph coloring (NP-hard)

1. Identify virtual registers that CANNOT share a physical register.
   1. Calculate live ranges
   2. Build a graph
2. Color the graph



```
Block 1:
    v1 := r4
    v2 := r5
    *v2 := 1
    v8 := v1 << 2
    v9 := v2 + v8
    *v9  := 1
    v17 := 1
    v42 := v1 >> 1
    v26 := v2
    v34 := v9
    v18 := v1
    v19 := v1
    v44 := v42 = 0
    if v44 goto Block 4
    goto Block 3
Block 2:
    v13 := v19 div v17
    v17 := v17 + 1
    v18 := v18 − 1
    v26 := v26 + 4
    v34 := v34 − 4
    v19 := v13 x v18
    *v26 := v13
    *v34 := v13
Block 3:
    v43 := v17 < v42
```



## Five things that I hope you will remember

- Abstraction: the separation of interface from implementation.
  - ISA's specify what the processor does, not how it does it.
- Locality:
  - Temporal Locality: "if you used it, you'll use it again"
  - Spatial Locality: "if you used it, you'll use something near it"
- Caching: buffering a subset of something nearby, for quicker access
  - Typically used to exploit locality.
- Indirection: adding a flexible mapping from names to things
  - Virtual memory's page table maps virtual to physical address.
- Throughput vs. Latency: (# things/time)  vs. (time to do one thing)
  - Improving one does not necessitate improving the other.

## Where to go from here?

- **CSE 401 Introduction to Compiler Construction (4)**
  **Fundamentals of compilers and interpreters; symbol tables; lexical analysis, syntax analysis, semantic analysis, code generation, and optimizations for general purpose programming languages. Prerequisite: CSE 322; CSE 326; CSE 341; CSE 378.**
- **CSE 451 Introduction to Operating Systems (4)**
  **Principles of operating systems. Process management, memory management, auxiliary storage management, resource allocation. Prerequisite: CSE 326; CSE 378.**
- **CSE 471 Computer Design and Organization (4)**
  **CPU instruction addressing models, CPU structure and functions, computer arithmetic and logic unit, register transfer level design, hardware and microprogram control, memory hierarchy design and organization, I/O and system components interconnection. Laboratory project involves design and simulation of an instruction set processor. Prerequisite: CSE 370; CSE 378.**
- **CSE 466 Software for Embedded Systems (4)**
  **Software issues in the design of embedded systems. Microcontroller architectures and peripherals, embedded operating systems and device drivers, compilers and debuggers, timer and interrupt systems, interfacing of devices, communications and networking. Emphasis on practical application of development platforms. Prerequisite: CSE 326; CSE 370; CSE 378.**