Lecture 6
Superscalar Elaboration, RISC, x86, and VAX

12 October 2006

Paper Topic

• Your paper topic is due today
  - I will take points off seriously late submissions

• By October 26th, please email me with no more than a paragraph telling what you plan to write about and cite a primary reference citation (please cite correctly)
Elaboration:
Superscalar, Tomasulo, Scoreboarding, and VLIW

Explanation of the Term "Superscalar"

• Definition:

  Superscalar machines are distinguished by their ability to (dynamically) issue multiple instructions each clock cycle from a conventional linear instruction stream

• In contrast to superscalar processors, VLIW processors use a long instruction word that contains a usually fixed number of instructions that are fetched, decoded, issued, and executed synchronously
Explaination of the Term "Superscalar"

- Instructions are issued from a sequential stream of normal instructions (in contrast to VLIW, in which a sequential stream of instruction tuples is used)
- The instructions that are issued are scheduled dynamically by the hardware (in contrast to VLIW processors, which rely on a static scheduling by the compiler)
- More than one instruction can be issued each cycle (motivating the term superscalar instead of scalar)
- The number of issued instructions is determined dynamically by the hardware, that is, the actual number of instructions issued in a single cycle can be zero up to a maximum instruction issue bandwidth (in contrast to VLIW where the number of scheduled instructions is fixed due to padding instructions with no-ops in case the full issue bandwidth would not be met)

Explaination of the Term "Superscalar"

- Dynamic issue of superscalar processors can allow issue of instructions either in-order or out-of-order
  - Only in-order issue is possible with VLIW processors
- The dynamic instruction issue complicates the hardware scheduler of a superscalar processor if compared with a VLIW
- The scheduler complexity increases when multiple instructions are issued out-of-order from a large instruction window
- It is a presumption of superscalar that multiple FUs are available
  - The number of available FUs is at least the maximum issue bandwidth, but often higher to diminish potential resource conflicts
Sections of a Superscalar Processor

- The ability to issue and execute instructions out-of-order partitions a superscalar pipeline in three distinct sections:
  - **In-order section** with the instruction fetch, decode and rename stages - the issue is also part of the in-order section in case of an in-order issue,
  - **Out-of-order** section starting with the issue in case of an out-of-order issue processor, the execution stage, and usually the completion stage, and again an
  - **In-order** section that comprises the retirement and write-back stages.

Multiple-issue Processors

- Scoreboarding and Tomasulo are the two single-issue techniques that allow out-of-order execution
- Most of today's general-purpose microprocessors are four- or six-issue superscalars, often with an enhanced Tomasulo scheme
- VLIW is the choice for most signal processors
- VLIW is used in Explicitly Parallel Instruction Computing (EPIC) by Intel in its IA-64 (Intel Architecture 64-bit) ISA—Itanium
Superscalar Pipeline with Tomasulo

- Instructions in the instruction window are free from control dependencies due to branch prediction, and free from name dependences due to register renaming.

- So, only (true) data dependences and structural conflicts remain to be handled.

- Meaning of: Issue, Dispatch, Commitment, Completion, Retirement

"Issue" (1/2)

- The issue logic examines the waiting instructions in the instruction window and simultaneously assigns (issues) a number of instructions to the FUs up to a maximum issue bandwidth.

- Several instructions can be issued simultaneously (the issue bandwidth).

- The program order of the issued instructions is stored in the reorder buffer.
“Issue” (2/2)

- Instruction issue from the instruction window can be:
  - In-order (only in program order) or out-of-order
  - It can be subject to simultaneous data dependences and resource constraints,
  - Or it can be divided in two (or more) stages
    » Checking structural conflict in the first and data dependences in the next stage (or vice versa)
    » In the case of structural conflicts first, the instructions are issued to reservation stations (buffers) in front of the FUs where the issued instructions await missing operands

Reservation Station(s)

- Be aware, there are two definitions in the literature:
  - A reservation station is a buffer for a single instruction with its operands (original Tomasulo paper, Flynn’s book, Hennessy/Patterson book)
  - A reservation station is a buffer (in front of one or more FUs) with one or more entries
    » Each entry can buffer an instruction with its operands (e.g. PowerPC literature)

- Depending on the specific processor, reservation stations can be central to a number of FUs or each FU has one or more own reservation stations
- Instructions await their operands in the reservation stations, as in the Tomasulo algorithm
“Dispatch”

- An instruction is said to be dispatched from a reservation station to the FU when all operands are available, and execution starts.
- If all its operands are available during issue and the FU is not busy, an instruction is immediately dispatched, starting execution in the next cycle after the issue.
- So, the dispatch is usually not a pipeline stage.
- An issued instruction may stay in the reservation station for zero to several cycles.
- Dispatch and execution is performed out of program order.
- Note: some authors interchange the meaning of issue and dispatch or use different semantics.

“Completion”

- When the FU finishes the execution of an instruction and the result is ready for forwarding and buffering, the instruction is said to complete.
- Instruction completion is out of program order.
- During completion the reservation station is freed and the state of the execution is noted in the reorder buffer.
- The state of the reorder buffer entry can denote an interrupt occurrence.
- The instruction can be completed and still be speculatively assigned, which is also monitored in the reorder buffer.
"Commitment"

• After completion, operations are committed in-order
• An instruction can be committed:
  - If all previous instructions due to the program order are already committed or can be committed in the same cycle,
  - If no interrupt occurred before and during instruction execution, and
  - If the instruction is no more on a speculative path
• By or after commitment, the result of an instruction is made permanent in the architectural register set, usually by writing the result back from the rename register to the architectural register

"Retirement"

• An instruction retires when the reorder buffer slot of an instruction is freed either
  - Because the instruction commits (the result is made permanent) or
  - Because the instruction is removed (without making permanent changes)
• A result is made permanent by copying the result value from the rename register to the architectural register
  - This is often done after the commitment of the instruction with the effect that the rename register is freed one cycle after commitment
RISC vs CISC

Major Advances in Computer Architecture (prior to RISC)

- Microprogrammed control unit
  - Idea by Wilkes 1951
  - Produced by IBM S/360 1964
- Cache memory
  - IBM S/360 model 85 1969
- Microprocessors
  - Intel 4004 (1971)
- Pipelining
  - Introduces parallelism into fetch execute cycle
- Multiple processors
- Solid State RAM
Design Characteristics of RISCs

- Simple instructions can be done in few clocks
  - Simplicity may even allow a shorter clock period
- A pipelined design can allow an instruction to complete in every clock period
- Fixed length instructions simplify fetch & decode
- The rules may allow starting next instruction without necessary results of the previous
  - Unconditionally executing the instruction after a branch
  - Starting next instruction before register load is complete

Design Characteristics of CISCs

- Design the machine to support the languages people actually use
  - They like functions, give them an instruction that really performs a function call
    » Transfers control, save “live” registers, etc.
    » Performs a virtual function call?
- If you are going to go get an instruction, make it complicated enough to give the processor something to do for a while
  - You don’t need to go out to memory as often
  - Your executable program is generally smaller
    » An IA32 binary tends to be about 20% smaller than MIPS.
Proponent's Claims

- **RISC**
  - Clearly superior, that is why they are covered in the text (and almost all other texts on architecture)
  - Load/Store architectures dominate new architectures

- **CISC**
  - Almost all processors used in desktop computers are CISC
  - Clearly the winner, because you probably use one (for now)

Classic Example

- **VAX Architecture: 1977-1992**
- **CISC instruction: CALLG**
  - Call procedure with general argument list
    - Transfer control (of course)
    - Point AP (argument pointer) register to arguments
    - Create stack frame (by decrementing Stack Pointer)
    - Saves registers specified to be saved (bit mask)
      - Bit mask is stored at target address (identifies which registers are used in the callee procedure)
      - First instruction of procedure is one word later
  - More about this architecture later
X86 (1)

- 16 bit registers (AX, BX, CX, DX), stack
- 0,1,2,3 address operand fields available
- Lots of addressing modes
- Powerful instructions, e.g., loop
  - CX must contain iteration count, conditional branch to target if decremented CX is not 0
- Next generation
  - Bigger registers (32 bit) - EAX, EBX, etc.
    » But remember to keep the old ones, because old instructions use them
  - No opcode space left?
    » Use one of the 256 possible opcodes as an escape (prefix)
      - Meaning, "this is not the opcode, the next byte(s) are"
      - And now I have a whole new 256+ unused opcodes

X86 (2)

- Next generation: graphics!
  - MMX - new data type
    
    | Red | Green | Blue | α |
    |-----|-------|------|---|
    | 0   | 7     | 8    | 15 |
    | 16  | 23    | 24   | 31 |
  - MMX - new instructions
    PADDUSB       opcode: 0x0F 0xDC
    Add unsigned bytes with saturation
Which Is Winning?

• Intel clearly can get their machines to run fast
• How?
  - By making the microarchitecture RISC-like and converting CISC to RISC during decode.

Examples (1)

• Pentium Pro microarchitecture
  - 12-14 cycle execution
  - Based on μops – RISC like instructions
    » Five cycle decode to translate CISC to RISC
    » Faster clock; higher CPI

• Pentium 4 microarchitecture
  - 20+ cycle execution (+ a lot sometimes)
  - Based on μops – RISC like instructions
    » Translate CISC to RISC (unspecified # of cycles)
    » Save it in the microarchitecture as RISC
    » Faster clock; higher CPI
Examples (2)

- History
  - IBM develops “Power” architecture
    » Basis of RS6000
    » Somewhere between RISC & CISC
  - IBM/Motorola/Apple combine to develop PowerPC architecture
    » Derivative of “Power” architecture used in Macintosh

- CISC-like features
  - Registers with control information
    » Set of condition registers (CR0–7) holding outcome of comparisons
    » Link register (LR) to hold return PC
    » Count register (CTR) to hold loop count
  - Updating load / stores
    » Update base register with effective address

CISC vs. RISC: Recap

- CISCs supply powerful instructions tailored to commonly used operations, stack operations, subroutine linkage, etc.
- RISCs require more instructions to do the same job
- CISC instructions take varying lengths of time
- RISC instructions can all be executed in the same, few cycle, pipeline
- RISCs should be able to finish (nearly) one instruction per clock cycle
Why CISC (1)?

- **Compiler simplification?**
  - Disputed...
  - Complex machine instructions harder to exploit
  - Optimization more difficult

- **Smaller programs?**
  - Program takes up less memory but...
  - Memory is now cheap
  - May not occupy less bits, just look shorter in symbolic form
    - More instructions require longer op-codes
    - Register references require fewer bits

Why CISC (2)?

- **Faster programs?**
  - Bias towards use of simpler instructions
  - More complex control unit
  - Microprogram control store larger
  - thus simple instructions take longer to execute

- **It is far from clear that CISC is the appropriate solution**
RISC vs. CISC (1)

- While a RISC machine may possibly have fewer instructions than a CISC, the instructions are always simpler. Multi-step arithmetic operations are confined to special units.
- Like all RISCs, the SPARC is a load/store machine. Arithmetic operates only on values in registers.
- A few, regular, instruction formats and limited addressing modes make instruction decode and operand determination fast.
- Branch delays are quite typical of RISC machines and arise from the way a pipeline processes branch instructions.

RISC vs. CISC (2)

- Not clear cut
- Many designs borrow from both philosophies, e.g., PowerPC and Pentiums
- Problems
  - No pair of RISC and CISC that are directly comparable
  - No definitive set of test programs
  - Difficult to separate hardware effects from compiler effects
  - Most comparisons done on “toy” rather than production machines
  - Most commercial devices are a mixture
Historic Personal Computers

Apple I 1976

IBM PC 1981

Atari 400 1979

Sinclair ZX80 1980

Apple Mac 1984
Quote...

- “DOS addresses only 1 MB of RAM because we cannot imagine any applications needing more.”
  - Microsoft, 1980

x86 Family (1)

- 1971: 4004
  - Intel’s first microprocessor
  - .1 MHz
- This is the engineering prototype of the Busicom calculator
- A CPU on-a-chip was developed as a standard product alternative to a set of custom circuits
- As market conditions in the calculator market worsened due to intense competition, Busicom renounced the exclusive rights for the 4-bit 4004 microprocessor for a better price, allowing Intel to introduce the 4004 to the general market in November 1971
x86 Family (2)

- **1972: 8008**
  - Twice as powerful as the 4004
  - Powered the Mark 8 home computer
- **1974: 8080**
  - Powered the first personal computer—the Altair
  - Computer hobbyists could purchase a kit for the Altair for $395. Within months, it sold tens of thousands, creating the first PC back orders in history.

x86 Family (3)

- **1978: 8086-8088 (PC and PC/XT)**
  - Powered IBM's new personal computer
  - 29,000 transistors
  - 5-10 MHz
  - Propelled Intel into the ranks of the Fortune 500, and Fortune magazine named the company one of the "Business Triumphs of the Seventies."
- **1982: 80286 (PC/AT)**
  - 134,000 transistors
  - 6-12.5 MHz
- **1985: 80386**
  - 275,000 transistors (more than 100 times as many as the original 4004)
  - 32-bit chip & external bus
  - Multi-tasking
x86 Family (4)

- **1989: 80486 DX**
  - First to offer a built-in math coprocessor
  - L1 cache on chip
  - 1.2 M transistors

- **1993: Pentium**
  - 3.1 M transistors

- **1995: Pentium Pro**
  - 5.5 M transistors
  - 64 bit bus

- **1997: Pentium II**
  - 7.5 M transistors
  - MMX technology, instruction set extension for video, audio and graphics data efficiently

x86 Family (5)

- **1999: Pentium III**
  - 70 new instructions (SIMD) for imaging, 3-D, streaming audio, and video
  - 9.5 million transistors
  - 0.25-micron technology.

- **2000: Pentium 4**
  - 42 million transistors
  - 0.18 microns
  - Initial speed of 1.5 GHz
Review Of MPU Fundamentals

- For Simplicity look at a simple model of an MPU
  - 8-bit
  - 64K address space
  - Intel-style interface

Simplified Block Diagram of a Microcomputer
Simple Microprocessor Model

MPU Model

Address Bus
16-bit
Uni-directional
Data Bus
8-bit
Bi-directional

I/O Read
I/O Write
Memory Read
Memory Write
Interrupt
Control

Processor Clock

Reset

DMA Control

Diagram of a Generic Microprocessor
General Registers

- Small set of internal registers - temporary data storage
- CU ensures that data from the correct register is presented to the ALU
- CU ensures that data is written back to correct register
- *Accumulator* usually holds ALU result

Status or Flags Register (Example)

O I T S Z A P C

- Overflow Flag
- Interrupt Flag
- Trap Flag
- Sign Flag
- Zero Flag
- Auxiliary Flag
- Parity Flag
- Carry Flag
Program Counter Register

- Points to the next register to be executed
- Called Instruction Pointer in Intel x86 Architecture

Stack Pointer

- STACK: Part of memory where program data can be stored by a simple PUSH operation
- Restore data by a POP
- Stack is in main memory and is defined by the program
- Stack Pointer (SP) keeps track of the next location available on the Stack
- Organized as a FILO Buffer
Memory Read and Write Cycles

- Hardware Control lines used by the CPU to Control reads and Writes to Memory
- Active low signal RD# asserted for a Read Cycle
- Active Low signal WR# indicates a write
- RD# and WR# signals supply timing information to memory device

Input and Output Cycles

- Intel Architecture processors have an I/O address space, separate from memory (Code and Data)
- Allow I/O devices to be decoded separately from memory devices
- Use IOR# and IOW# signals for Input & Output
- Exercise: Draw Input & Output Cycles following the memory cycle examples
I/O Instructions

- Separate I/O instructions cause the IOR# or IOW# signals to be asserted
  - MOV AL, (400Fh) ; instruction provides 16-bit address
  - IN AL, 2Ch ; instruction provides an 8-bit address

- Some processors only support a single address space - I/O devices are decoded in the memory map

Advantages of Memory Mapped I/O

- I/O locations are read/written by normal instructions - no need for separate I/O instructions
  - Size of instruction set reduced
- Memory manipulations can be performed directly on I/O locations
- No need for IOR# and IOW# pins
Advantages of Separate I/O Mapping

- All locations in memory map are available for memory
  - No block removed for I/O
- Smaller, faster instructions can be used for I/O
- Less Hardware decoding for I/O
- Easier to distinguish I/O accesses in assembly language
- Which mapping system is preferable? Why?
Device Selection and Data Buses (1)

- A PC board may have many memory devices, all attached to the same data bus
- When the processor reads data from the bus, it is essential that only one device drives data onto the bus
- The other memories must be electrically disconnected from the bus while the selected device drives it

Device Selection and Data Buses (2)

- Use Address Decoding to ensure only one device is selected at a time
- Use Tristate buffers to disconnect unselected devices from the data bus
- Unselected devices have their outputs placed in the HIGH IMPEDANCE STATE – it's as if their outputs were switched off
- All outputs, except those of the selected device should be in the High Impedance state
Interrupts

- Used to Halt the normal flow of instructions
- Exceptions can be due to Hardware or Software
- Hardware Interrupts are asynchronous to the processor
- Could be asserted by an external device requesting action, e.g. a port ready to transfer data
- Interrupts can be globally masked by the processor’s Interrupt Enable Flag (IE or I)
- IE is set by STI and reset by CLI (or equivalent)

Maskable & Non Maskable Interrupts

- Maskable interrupts can be enabled/disabled using a flag (usually in the flags register)
- Non Maskable Interrupts (NMI) are top priority interrupts that can’t be masked out
- NMIs often used for Parity Errors, Power fails etc
Interrupts

Operations shown in boxes are carried automatically by MPU hardware

Direct Memory Access (DMA)

- DMA techniques improve system performance
- External devices can transfer data directly to or from memory under hardware control
- Other methods (e.g. interrupts) use software to transfer data and are slower
- DMA is used when very high data rates are required
• **Pinnacle of CISC ! ! !**
  - Maximize instruction density
  - Provide instructions closely matched to typical program operations

• **Instruction format**
  - OP, arg1, arg2, ...
    - Each argument has arbitrary specifier
    - Accessing operands may have side effects

• **Condition Codes**
  - Set by arithmetic and comparison instructions
  - Basis for successive branches

• **Procedure Linkage**
  - Direct implementation of stack discipline
VAX Architecture (1)

- **General register machine**: 16 32-bit registers
- **Special register assignments**:
  - R15: program counter
  - R14: current stack pointer
  - R13: stack frame pointer
  - R12: argument list pointer
  - R0-R5: state information for long interruptible instructions
- **32-bit addressing**

VAX Architecture (2)

- **Superset of many other ISAs**, including IBM 360/370 and DEC PDP-11.
- **VAX** = “Virtual Address Extension: solve the problem of 16-bit
- **Addressing on the PDP-11**, but be compatible:
  - Same data types
  - Same I/O bus (UNIBUS) plus others (MASSBUS, SBI)
  - PDP-11 direct emulation mode
  - Similar assembler syntax
- **Large orthogonal instruction set**, with support for:
  - High-level language constructs (procedure call w/ arguments)
  - Virtual memory management
  - Rapid context switching (Process Control Block: Regs+state)
- **First implementation**: the VAX-11/780 (~1976)
  - Internally microprogrammed, 96-bit word + writeable section
VAX Registers

- R0-R11 General purpose
  - Use pair to hold double
- R2-R7 Callee save in example code
  - R2-R7 Callee save in example code
- R12 AP Argument pointer
  - Stack region holding procedure arguments
- R13 FP Frame pointer
  - Base of current stack frame
- R14 SP Stack pointer
  - Top of stack
- R15 PC Program counter
  - Used to access data in code
- N C V Z Condition codes
  - Information about last operation result
  - Negative, Carry, 2’s OVF, Zero

VAX Instruction Encoding

- 1-2 byte opcodes followed by 0-6 operand specifiers, each of which may be up to 5 bytes.
- About 300 opcodes, most 8-bit (integer, floating point, character string, bit fields, decimal, CRC, POLY)
- Opcode implies data type and size, and number of operands (A=B+C vs A=A+B)
- Operand specifiers indicate addressing modes
- Orthogonality: any opcode with any operands of any modes
- How do you encode?
VAX Operand Addressing

- 6-bit Literal: 0–63 positive integers, 3-bit exp+3-bit fraction floating point (integers 1–16, 16ths, more)
- Register, register deferred, optional autoincrement/decrement
- 1, 2, 4-byte displacement from register, optionally deferred
- Indexed (scaled): R* operand-size + next operand specifier (for array indexing)
- Clever encoding:
  - immediate = PC autoincrement
  - absolute = PC autoincrement deferred

Summary

- ISAs are shaped by implementations, compilers, and history
  - Limited hardware + primitive compilers --> stack architectures
  - HLL support + high memory cost --> orthogonal CISC
  - Limited hardware + much assembly programming --> small register set designs
  - Efficient pipelining + optimizing compilers + good caches --> RISC architectures
- What's coming?
  - More support for multiple instruction issue
  - Increased tolerance for long memory latency
  - Continued “complexification” of RISC, within reason
  - Even tighter compiler/ISA integration; compiler-generated hints for
  - branches, memory accesses, etc.
  - Support for multiple threads of control