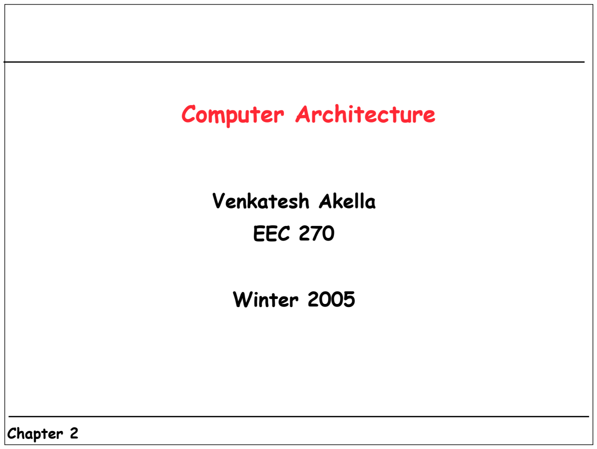
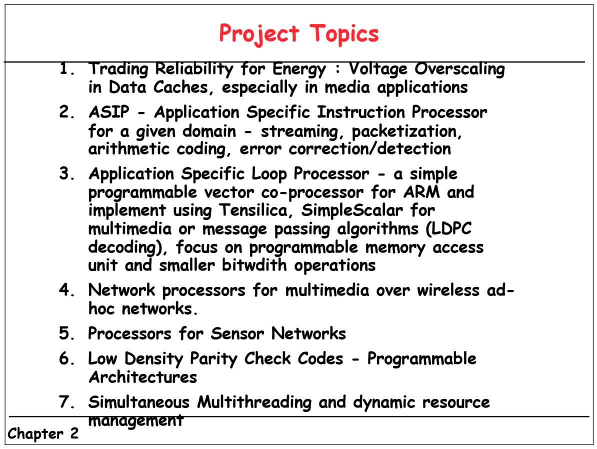


Computer Architecture

Venkatesh Akella

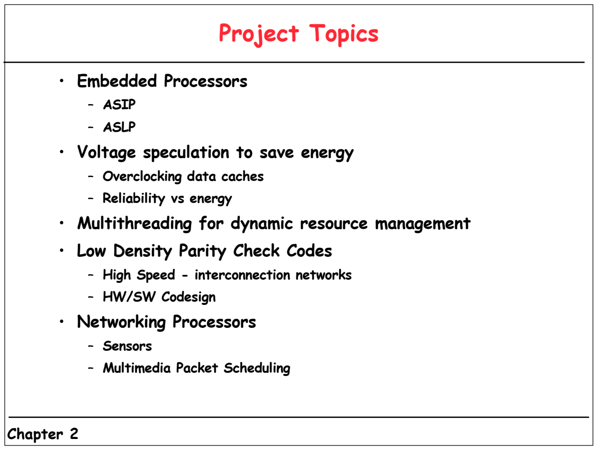
EEC 270

Winter 2005



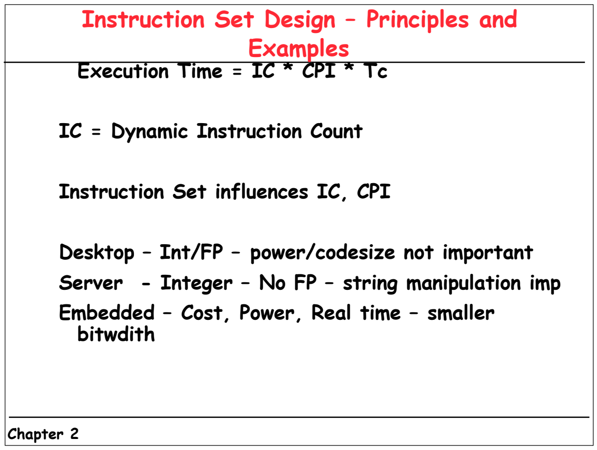
Project Topics

1. Trading Reliability for Energy : Voltage Overscaling in Data Caches, especially in media applications
2. ASIP - Application Specific Instruction Processor for a given domain - streaming, packetization, arithmetic coding, error correction/detection
3. Application Specific Loop Processor - a simple programmable vector co-processor for ARM and implement using Tensilica, SimpleScalar for multimedia or message passing algorithms (LDPC decoding), focus on programmable memory access unit and smaller bitwdith operations
4. Network processors for multimedia over wireless ad-hoc networks.
5. Processors for Sensor Networks
6. Low Density Parity Check Codes - Programmable Architectures
7. Simultaneous Multithreading and dynamic resource management



Project Topics

- Embedded Processors
  - ASIP
  - ASLP
- Voltage speculation to save energy
  - Overclocking data caches
  - Reliability vs energy
- Multithreading for dynamic resource management
- Low Density Parity Check Codes
  - High Speed - interconnection networks
  - HW/SW Codesign
- Networking Processors
  - Sensors
  - Multimedia Packet Scheduling



Instruction Set Design – Principles and Examples

Execution Time = IC * CPI * Tc

IC = Dynamic Instruction Count

Instruction Set influences IC, CPI

Desktop – Int/FP – power/codesize not important

Server - Integer – No FP – string manipulation imp

Embedded – Cost, Power, Real time – smaller bitwdith

Overview of the Chapter

- What are the alternatives and trade-offs?
- Taxonomy of ISA and quantitative assessment
- ISA of embedded and DSP processors
- Role of Compilers and High-level Languages
- TriMedia and MIPS64 Cases Study

Anatomy of an Instruction

- Operation
  - Arithmetic
  - Logical
  - Control Flow
  - Procedure Call
- Operands
  - Type of operands (bit, byte, char, string, float, int)
  - Addressing the operands (Addressing modes)
- Representation in the memory (encoding)
  - Fixed vs variable
  - Aligned vs unaligned
  - Lilliputian Wars
  - Compressed vs Uncompressed
  - Impacts code size, decode efficiency,

Classifying Instruction Set Architectures

Machine Type	Advantages	Disadvantages
Stack	Simple effective address Short instructions Good code density Simple I-decode	Lack of random access. Efficient code is difficult to generate. Stack is often a bottleneck.
Accumulator	Minimal internal state Fast context switch Short instructions Simple I-decode	Very high memory traffic
Register	Lots of code generation options. Efficient code since compiler has numerous useful options.	Longer instructions. Possibly complex effective address generation. Size and structure of register set has many options.

Why did Register-Register ISA survive?

- Registers are faster than memory
- Take advantage of principle of locality
- Flexibility – Consider the expression:

(A*B) – (B*C) – (A*D)

There are several ways of evaluating this on a R-R machine but on a stack based machine it is restricted to one order

. Code density – registers can be specified with fewer bits than memory addresses

. Reduces memory traffic – locality

. Amenable for automatic compilation

Addressing Modes - Note the relationship to high-level language constructs

Mode	Example Instruction	Meaning	Use
Register	Add R4, R3	Regs[R4] <- Regs[R4] + Regs[R3]	All RISC ALU operations
Immediate	Add R4, #3	Regs[R4] <- Regs[R4] + 3	for small constants - problems?
Displacement	Add R4, 100(R1)	Regs[R4] <- Regs[R4] + Mem[100 + Regs[R1]]	accessing local variables
Register deferred or Indirect	Add R4, (R1)	Regs[R4] <- Regs[R4] + Mem[Regs[R1]]	pointers
Indexed	Add R3, (R1 + R2)	Regs[R3] <- Regs [R3] + Mem[Regs[R1] + Regs[R2]]	array access - R1 is the base, R2 is the index
Direct or absolute	Add R1, (1001)	Regs[R1] <- Regs[R1] + Mem[1001]	problems?

More Addressing Modes

Mode	Example Instruction	Meaning	Use
Memory Indirect or Memory Deferred	Add R1, @R3	Regs[R1] <- Regs[R1] + Mem[Mem[Regs[3]]]	If R3 holds a pointer address, then result is the full dereferenced pointer

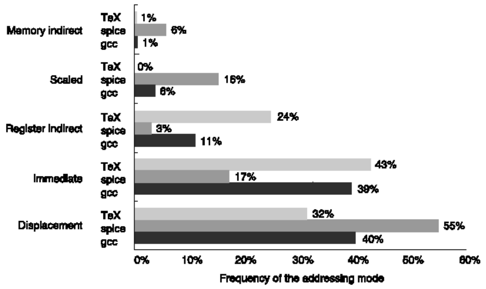
Autoincrement
in this case post increment
note symmetry with autodec
	Add R1, (R2) +	Regs[R1] <- Regs[R1] + Mem[Regs[R2]];
		Regs[R2] <- Regs[R2] + d
			Array walks - if element of size d is accessed then pointer increments auto

Autodecrement
in this case predecrement
	Add R1, - (R2)	Regs[R2] <- Regs[R2] - d;
		Regs[R1] <- Regs[R1] + Mem[Regs[R2]];
			array walks, with autoinc useful for stack implementation

Scaled	Add R1, 100 (R2) [R3]	Regs[R1] <- Regs[R1] + Mem[100 + Regs[R2] + Regs[R3] * d]	array access - may be applied to indexed addressing in some machines

d ::= size of an element

Use of Memory Addressing Modes



Vax Measurements based on SPEC89 benchmarks

X86 Measurements

Rank	x86 instruction	% of total instructions
1	load	22%
2	conditional branch	20%
3	compare`	16%
4	sstore	12%
5	add	8%
6	and	6%
7	sub	5%
8	move reg-reg	4%
9	call	1%
10	return	1%
TOTAL		96%

Small register file bloats number of loads and stores

Mix of Instructions on TI DSP C54x

Instruction	Percent
Store mem16	32.2%
Load mem16	9.4%
Add mem16	6.8%
CALL	5%
Push mem16	5%
Subtract mem16	4.9%
Move mem-mem16	4.0%
MAC	4.6%

Interesting Questions

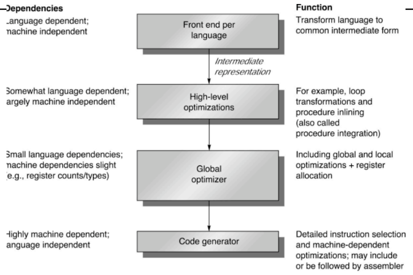
- How many bits for the immediate field?
- How many registers do I need?
- What addressing modes to support?
- What operations to support?

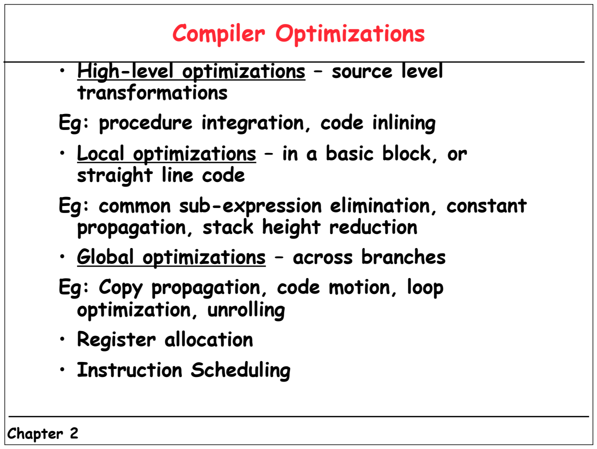
- Amadahl's law
- Make the common case fast
- Can the compiler use it?
- Don't forget, ultimately Execution Time matters – so always look at the impact on IC, CPI and Tc
- Measure on Real Benchmarks!

Role of a Compiler

- Today majority of programming is in high-level languages.
- So, the Instruction set should be amenable for a compiler as a target
- Case in point – DSP, micro controllers are not which makes software development a nightmare
- Remember – architecture is a codesign issue, so a smart compiler can help if the architecture exposes some aspects
- Eg: instruction scheduling to avoid pipeline stalls, hide long latency of memory, use the registers better, avoid recomputation, code size optimization, cache optimization, …….

Anatomy of a compiler


Dependencies
Language dependent; machine independent
Somewhat language dependent; largely machine independent
Small language dependencies; machine dependencies slight (e.g., register counts/types)
Highly machine dependent; language independent
Front end per language
Intermediate representation
High-level optimizations
Global optimizer
Code generator
Function
Transform language to common intermediate form
For example, loop transformations and procedure inlining (also called procedure integration)
Including global and local optimizations + register allocation
Detailed instruction selection and machine-dependent optimizations; may include or be followed by assembler



Compiler Optimizations

- High-level optimizations – source level transformations

Eg: procedure integration, code inlining

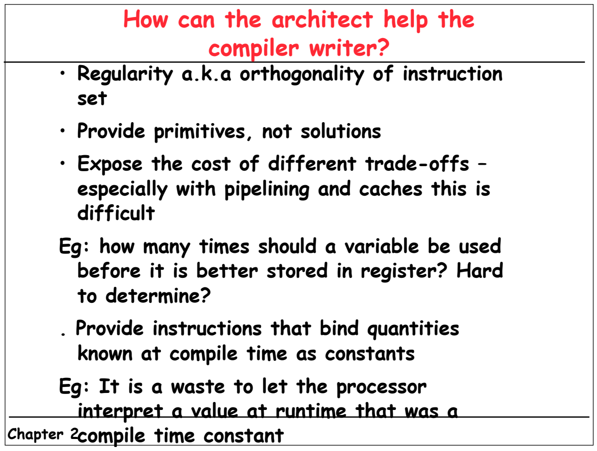
- Local optimizations – in a basic block, or straight line code

Eg: common sub-expression elimination, constant propagation, stack height reduction

- Global optimizations – across branches

Eg: Copy propagation, code motion, loop optimization, unrolling

- Register allocation
- Instruction Scheduling



How can the architect help the compiler writer?

- Regularity a.k.a orthogonality of instruction set
- Provide primitives, not solutions
- Expose the cost of different trade-offs – especially with pipelining and caches this is difficult

Eg: how many times should a variable be used before it is better stored in register? Hard to determine?

. Provide instructions that bind quantities known at compile time as constants

Eg: It is a waste to let the processor interpret a value at runtime that was a compile time constant