Instruction Set Architecture and its Implications

A.   Moshovos (c)

Based on H&P CA: AQA
Some Figures and Tables taken directly from the book
These are not meant to be slides but notes

Overview

- Instruction Set Architecture Overview

- Interaction of ISA and compilers

- Interaction of ISA and implementation

- Key Idea:

    - The ISA can make the compiler or the implementation harder

    - Some may argue that the ISA can have a profound impact on performance, cost and complexity
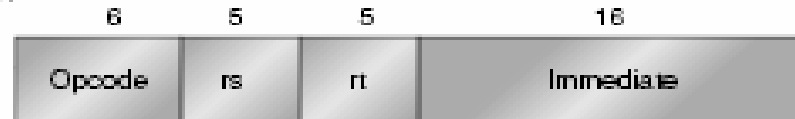
Instruction Set Architecture: Definition

- The set of all instructions

- Their format (binary representation)

- Specification of their operation

- Includes Machine state such as

    - Type of Operands

    - Memory address space

MIPS ISA Formats

I-type instruction

6	5	5	16
Opcode	rs	rt	Immediate

Encodes: Loads and stores of bytes, half words, words, double words. All immediates (rt —rs op immediate)

Conditional branch instructions (rs is register, rd unused)
Jump register, jump and link register
        (rd = 0, rs = destination, immediate = 0)

R-type instruction

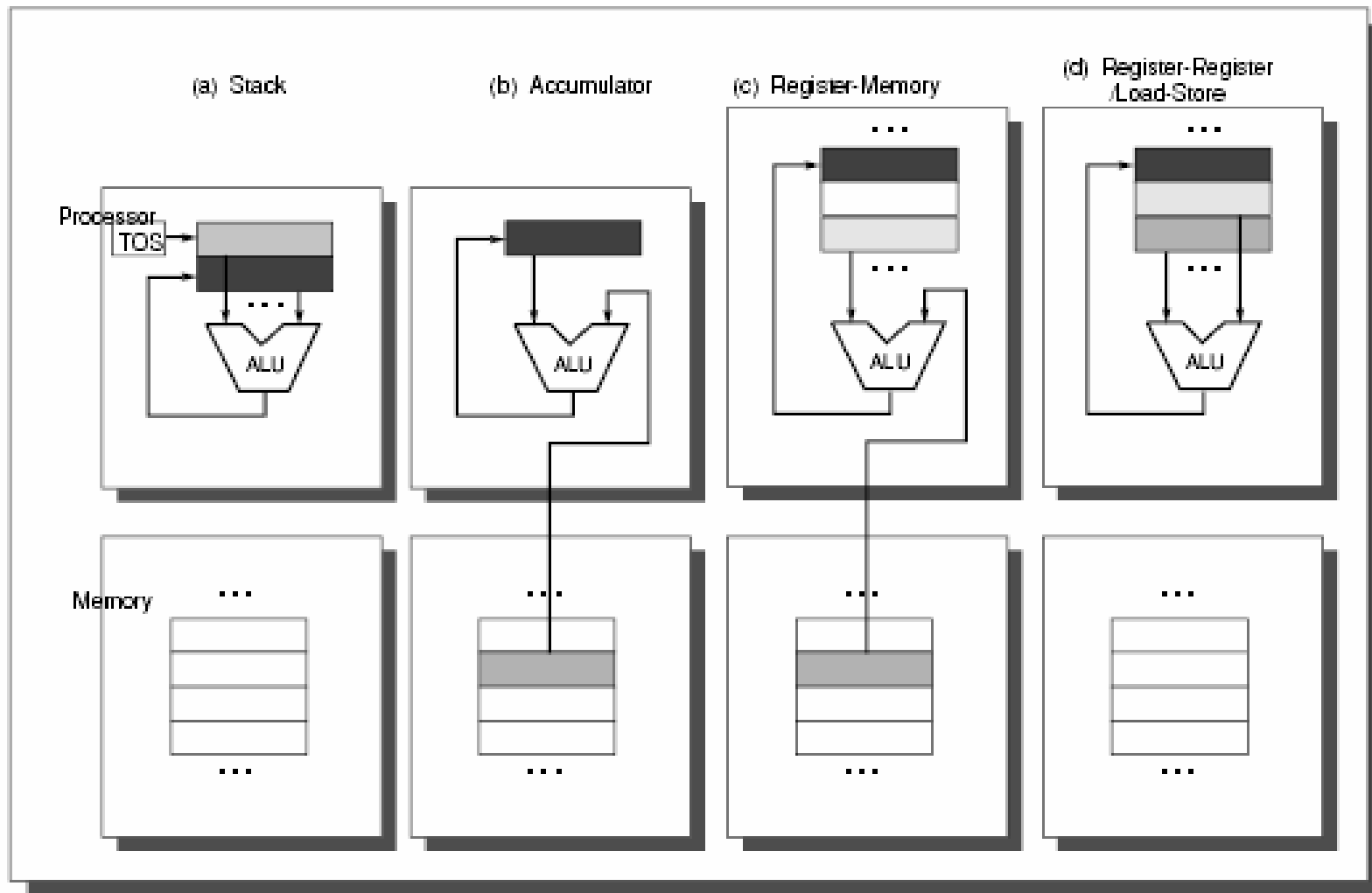| 6	5	5	5	5	6
Opcode	rs	rt	rd	shamt	funct

Register—register ALU operations: rd— rs funct rt
        Function encodes the data path operation: Add, Sub, . . .
        Read/write special registers and moves

J-type instruction

6	26
Opcode	Offset added to PC

Jump and jump and link
Trap and return from exception

Classifying Instruction Sets based on Operand Location

C = A + B

Stack	Accumulator	Register (register-memory)	Register (load-store)
Push A	Load A	Load R1,A	Load R1,A
Push B	Add B	Add R3,R1,B	Load R2,B
Add	Store C	Store R3,C	Add R3,R1,R2
Pop C			Store R3,C

Comparing Instruction Sets based on Operand Location

Type	Advantages	Disadvantages
Register-register (0,3)	Simple, fixed-length instruction encoding. Simple code-generation model. Instructions take similar numbers of clocks to execute (see App. A).	Higher instruction count than architectures with memory references in instructions. More instructions and lower instruction density leads to larger programs.
Register-memory (1,2)	Data can be accessed without a separate load instruction first. Instruction format tends to be easy to encode and yields good density.	Operands are not equivalent since a source operand in a binary operation is destroyed. Encoding a register number and a memory address in each instruction may restrict the number of registers. Clocks per instruction vary by operand location.
Memory-memory (2,2) or (3,3)	Most compact. Doesn't waste registers for temporaries.	Large variation in instruction size, especially for three-operand instructions. In addition, large variation in work per instruction. Memory accesses create memory bottleneck. (Not used today.)

FIGURE 2.4   Advantages and disadvantages of the three most common types of general-purpose register computers. The notation (m, n) means m memory operands and n total operands. In general, computers with fewer alternatives simplify the compiler's task since there are fewer decisions for the compiler to make (see section 2.11). Computers with a wide variety of flexible instruction formats reduce the number of bits required to encode the program. The number of registers also affects the instruction size since you need $\log_2$ (number of registers) for each register specifier in an instruction. Thus, doubling the number of registers takes 3 extra bits for a register-register architecture, or about 10% of a 32-bit instruction.

Addressing Modes

Addressing mode	Example instruction	Meaning	When used
Register	Add R4,R3	Regs[R4]←Regs[R4] + Regs[R3]	When a value is in a register.
Immediate	Add R4,#3	Regs[R4]←Regs[R4]+3	For constants.
Displacement	Add R4,100(R1)	Regs[R4]←Regs[R4] + Mem[100+Regs[R1]]	Accessing local variables (+ simulates register indirect, direct addressing modes)
Register indirect	Add R4,(R1)	Regs[R4]←Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer or a computed address.
Indexed	Add R3,(R1 + R2)	Regs[R3]←Regs[R3] +Mem[Regs[R1]+Regs[R2]]	Sometimes useful in array addressing: R1 = base of array; R2 = index amount.
Direct or absolute	Add R1,(1001)	Regs[R1]←Regs[R1] + Mem[1001]	Sometimes useful for accessing static data; address constant may need to be large.
Memory indirect	Add R1,@(R3)	Regs[R1]←Regs[R1] + Mem[Mem[Regs[R3]]]	If R3 is the address of a pointer p, then mode yields $*p$.
Autoincrement	Add R1,(R2)+	Regs[R1]←Regs[R1] + Mem[Regs[R2]] Regs[R2]←Regs[R2]+d	Useful for stepping through arrays within a loop. R2 points to start of array; each reference increments R2 by size of an element, d.
Autodecrement	Add R1,-(R2)	Regs[R2]←Regs[R2]-d Regs[R1]←Regs[R1] + Mem[Regs[R2]]	Same use as autoincrement. Autodecrement/increment can also act as push/pop to implement a stack.
Scaled	Add R1,100(R2)[R3]	Regs[R1]← Regs[R1]+ Mem[100+Regs[R2] + Regs[R3]*d]	Used to index arrays. May be applied to any indexed addressing mode in some computers.

Do we need all these addressing modes?
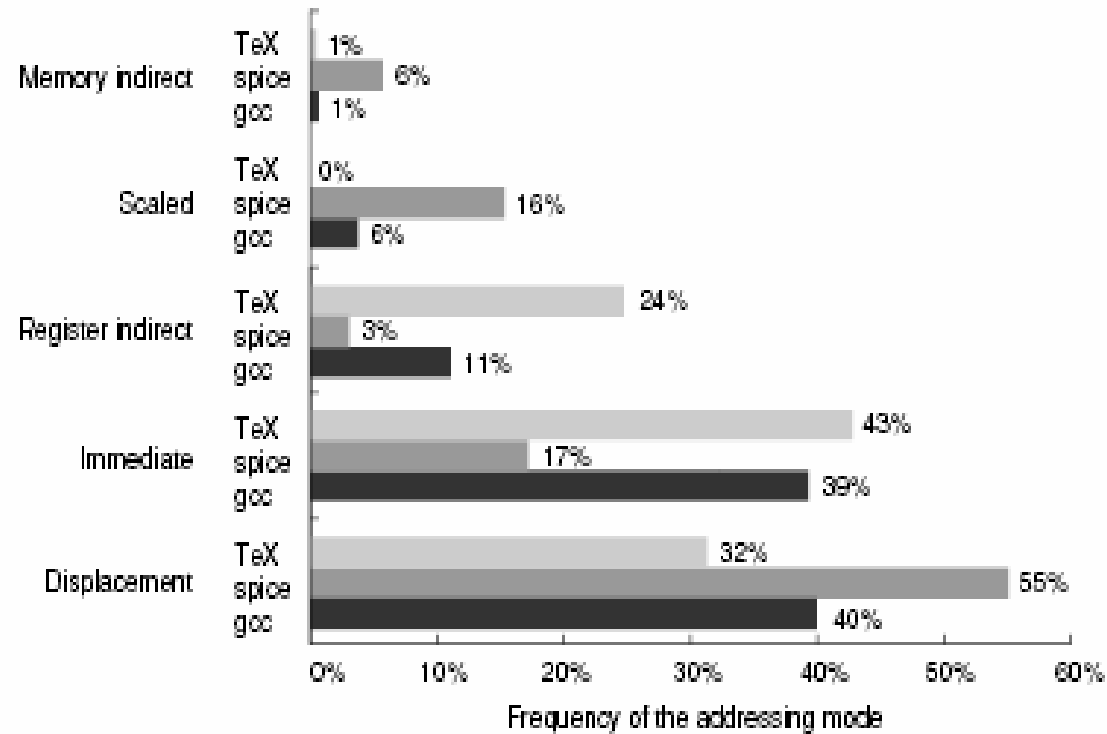
Immediate, Displacement and Register Indirect dominate
Displacement Constants vary but most are small < 16bit

Operations

Operator type	Examples
Arithmetic and logical	Integer arithmetic and logical operations: add, subtract, and, or, multiple, divide
Data transfer	Loads-stores (move instructions on computers with memory addressing)
Control	Branch, jump, procedure call and return, traps
System	Operating system call, virtual memory management instructions
Floating point	Floating-point operations: add, multiply, divide, compare
Decimal	Decimal add, decimal multiply, decimal-to-character conversions
String	String move, string compare, string search
Graphics	Pixel and vertex operations, compression/decompression operations

FIGURE 2.15   Categories of instruction operators and examples of each. All computers generally provide a full set of operations for the first three categories. The support for system functions in the instruction set varies widely among architectures, but all computers must have some instruction support for basic system functions. The amount of support in the instruction set for the last four categories may vary from none to an extensive set of special instructions. Floating-point instructions will be provided in any computer that is intended for use in an application that makes much use of floating point. These instructions are sometimes part of an optional instruction set. Decimal and string instructions are sometimes primitives, as in the VAX or the IBM 360, or may be synthesized by the compiler from simpler instructions. Graphics instructions typically operate on many smaller data items in parallel; for example, performing eight 8-bit additions on two 64-bit operands.

Common Operations: x86 – Spec92

Rank	80x86 instruction	Integer average (% total executed)
1	load	22%
2	conditional branch	20%
3	compare	16%
4	store	12%
5	add	8%
6	and	6%
7	sub	5%
8	move register-register	4%
9	call	1%
10	return	1%
Total		96%

FIGURE 2.16   The top 10 instructions for the 80x86. Simple instructions dominate this list, and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs.

Control Flow Instructions

- Conditional Branches

- Jumps

  - Unconditional Branches

- Procedure Calls

- Procedure Returns

Conditional Control Flow Operations

Name	Examples	How condition is tested	Advantages	Disadvantages
Condition code (CC)	80x86, ARM, PowerPC, SPARC, SuperH	Special bits are set by ALU operations, possibly under program control.	Sometimes condition is set for free.	CC is extra state. Condition codes constrain the ordering of instructions since they pass information from one instruction to a branch.
Condition register	Alpha, MIPS	Tests arbitrary register with the result of a comparison.	Simple.	Uses up a register.
Compare and branch	PA-RISC, VAX	Compare is part of the branch. Often compare is limited to subset.	One instruction rather than two for a branch.	May be too much work per instruction for pipelined execution.

FIGURE 2.21 The major methods for evaluating branch conditions, their advantages, and their disadvantages. Although condition codes can be set by ALU operations that are needed for other purposes, measurements on programs show that this rarely happens. The major implementation problems with condition codes arise when the condition code is set by a large or haphazardly chosen subset of the instructions, rather than being controlled by a bit in the instruction. Computers with compare and branch often limit the set of compares and use a condition register for more complex compares. Often, different techniques are used for branches based on floating-point comparison versus those based on integer comparison. This dichotomy is reasonable since the number of branches that depend on floating-point comparisons is much smaller than the number depending on integer comparisons.

Encoding Instruction Sets

- Balance amongst competing forces

- As many registers as possible

- Impact on instruction size and program size

- Simplicity of decoding
  - Multiple of bits, bytes or fixed size?

Basic Variations in Instruction Encoding

Operation & no. of operands	Address specifier 1	Address field 1	· · ·	Address specifier n	Address field n

(a) Variable (e.g., VAX, Intel 80x86)

Operation	Address field 1	Address field 2	Address field 3

(b) Fixed (e.g., Alpha, ARM, MIPS, PowerPC, SPARC, SuperH)

Operation	Address specifier	Address field

Operation	Address specifier 1	Address specifier 2	Address field

Operation	Address specifier	Address field 1	Address field 2

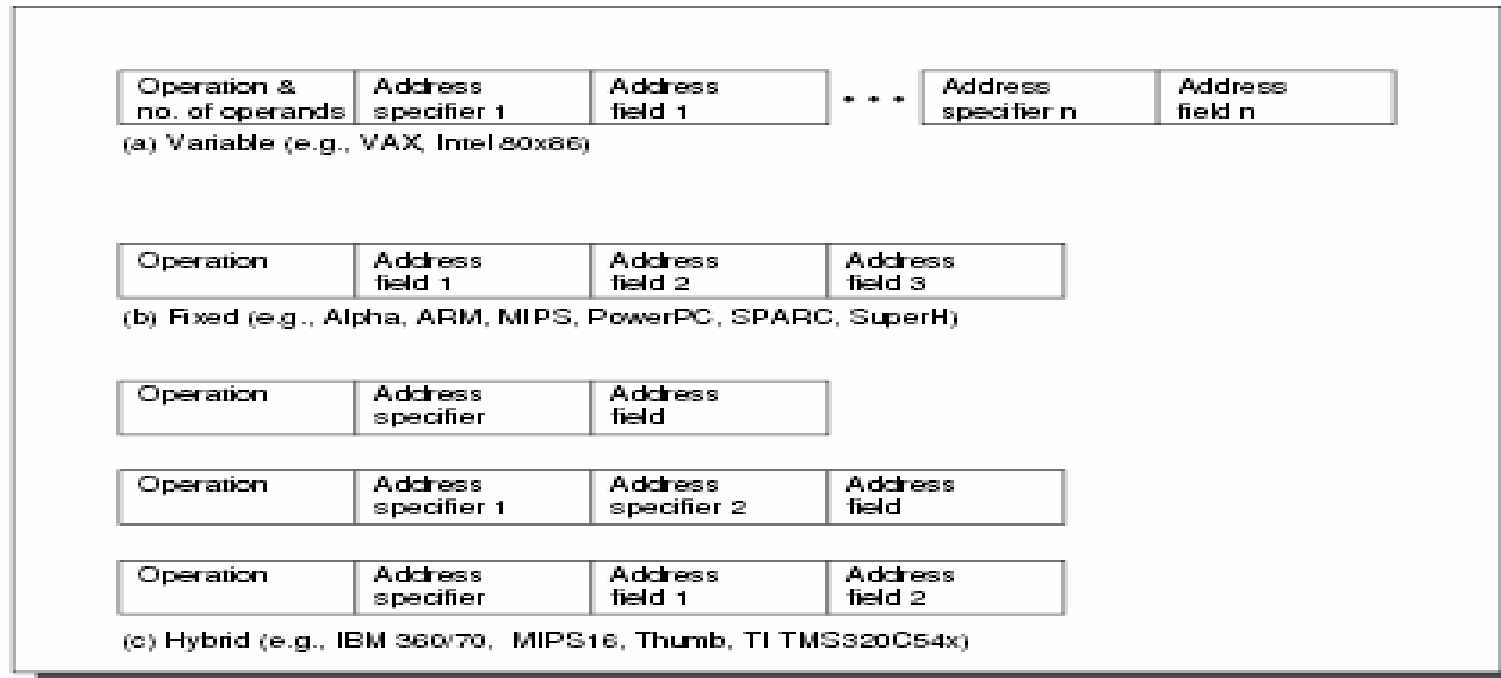(c) Hybrid (e.g., IBM 360/70, MIPS16, Thumb, TI TMS320C54x)

FIGURE 2.23   Three basic variations in instruction encoding: variable length, fixed length, and hybrid. The variable format can support any number of operands, with each address specifier determining the addressing mode and the length of the specifier for that operand. It generally enables the smallest code representation, since unused fields need not be included. The fixed format always has the same number of operands, with the addressing modes (if options exist) specified as part of the opcode (see also Figure C.3 on page C-4). It generally results in the largest code size. Although the fields tend not to vary in their location, they will be used for different purposes by different instructions. The hybrid approach has multiple formats specified by the opcode, adding one or two fields to specify the addressing mode and one or two fields to specify the operand address (see also Figure D.7 on page D-12).

Compilers and the ISA

- Compilers will be used to generate most programs
- Goal of the Compiler
  - Generate the best code
  - Best: size and/or speed
- The compiler solves a big optimization problem
- The ISA can make this harder or easier

Structure of a Modern Optimizing Compiler

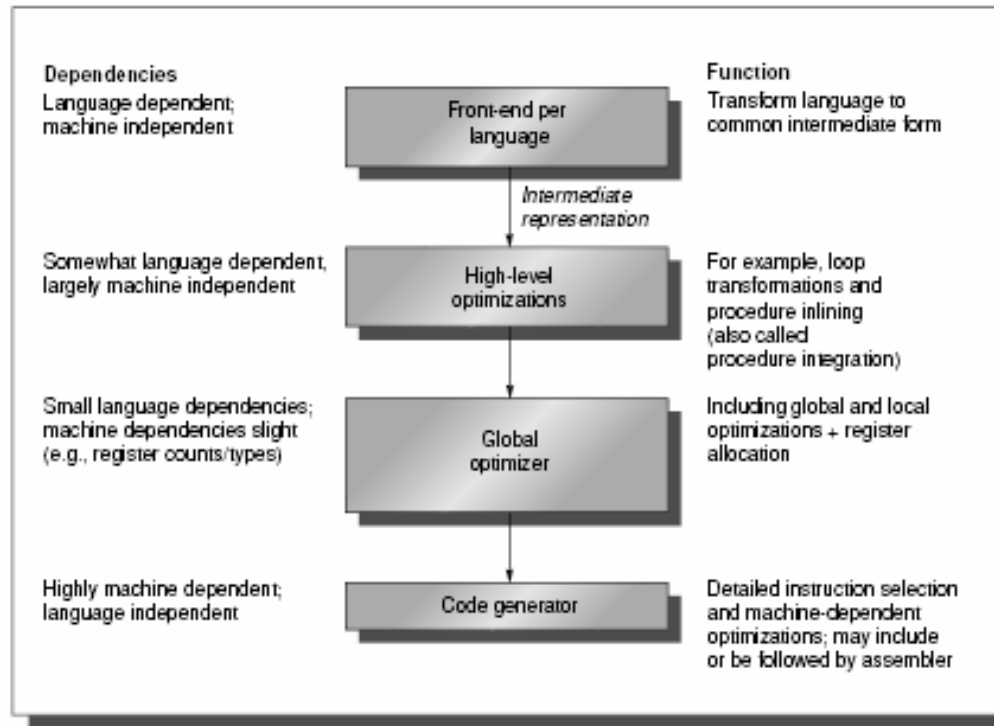| Dependencies		Function
Language dependent; machine independent	Front-end per language	Transform language to common intermediate form
	Intermediate representation	
Somewhat language dependent, largely machine independent	High-level optimizations	For example, loop transformations and procedure inlining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

FIGURE 2.24   Compilers typically consist of two to four passes, with more highly optimizing compilers having more passes. This structure maximizes the probability that a program compiled at various levels of optimization will produce the same output when given the same input. The optimizing passes are designed to be optional and may be skipped when faster compilation is the goal and lower quality code is acceptable. A *pass* is simply one phase in which the compiler reads and transforms the entire program. (The term *phase* is often used interchangeably with *pass*.) Because the optimizing passes are separated, multiple languages can use the same optimizing and code-generation passes. Only a new front end is required for a new language.

How the ISA can help the Compiler

- Goal:
  - Make Frequent Cases fast and the Infrequent Correct

- Regularity
  - Operation, data types and addressing modes should be <u>orthogonal</u>
    - Can use any combination
    - Counter-example: special purpose registers (sp and bp in x86)

- Provide primitives not solutions
  - Attempting to support high-level programming language features may result in suboptimal results
    - Example function prologue/epilogue saving/restoring instructions

How the ISA can help the Compiler

- Simplify tradeoffs among alternatives
  - What is the best instruction sequence given an operation?
    - Instruction count is not a good metric
    - Think of a register-memory architecture like x86
      - At which point it makes sense to register allocate a variable?