Encoding an Instruction

- The previous design decisions (e.g. bits of branch displacement) influence the instruction encoding.
- It determents the size of the program and the implementation of the processor.
- The operation is encode in the opcode field.
- The important decision is how to encode the addressing modes with the operation.
- ISA with many addressing modes require address specifier fields to associate addressing modes with the operand.

Addressing Modes[1]

Addressing mode	Example instruction	Register transfer	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 and simulates register indirect/direct addressing
Register indirect	Add R4,(R1)	Regs[R4]<- Regs[R4] + Mem[Regs[R1]]	Accessing using a pointer
Index	Add R3,(R1+R2)	Regs[R3]<- Regs[R3] + Mem[Regs[R1]+Regs[R2]]	Useful in array addressing

Addressing Modes[2]

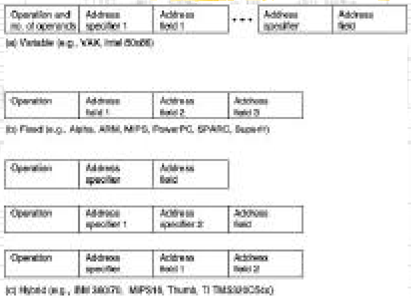
Addressing mode	Example instruction	Register transfer	When used
Direct or absolute	Add R1,(1001)	Regs[R1]<- Mem[1001]	Useful to access static data.
Memory indirect	Add R1,@(R3)	Regs[R1]<- Mem[MEM[Regs[R3]]]	If R3 is the address of a pointer p, then mode yields *p.
Autoincrement	Add R1,(R2)+	Regs[R1]<- Mem[Mem[Regs[R2]] Regs[R2]<- Regs[R2]+d	Stepping through arrays.
Autodecrement	Add R1,-(R2)	Regs[R2]<- Regs[R2]-d Regs[R1]<- Mem[Mem[Regs[R2]]	Stepping through arrays.
Scaled	Add R1,100(R2)[R3]	Regs[R1]<-Regs[R1] + Mem[100+Regs[R2]+Regs[R3]*d]	Used to index arrays.

ISA Encoding

- Load-store ISAs can encode the addressing mode in the opcode.
- The number of registers and number of addressing modes both have a significant impact on the size of the instruction.
- The ISA should balance:
  - Number of registers and addressing modes
  - The impact on the size of the instruction (-> program)
  - Size that can be efficient implemented.

Variations in Instruction Encoding

Figure 2.23 - Hennessy & Patterson



80x86 and RISC

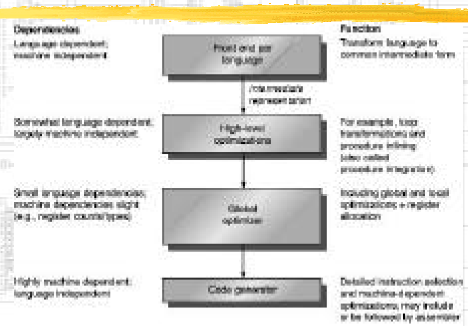
- The 80x86 instructions vary between 1 and 17 bytes.
- These programs are smaller than RISC architectures.
- Some RISC architectures (ARM Thumb and MIPS 16) reduce the code size by up to 40% for embedded application through:
  - Narrow instructions
  - Fewer operations
  - Smaller addresses
  - Smaller immediate fields
  - Fewer registers
  - Two-address formats and not three
- Some architectures use compression.

ISA and Compilers

- The ISA is a compiler target.
- The compiler significantly affects the performance.
- Understanding of compilers is necessary to implement an efficient instruction set.
- The figure on the following slide shows the structure of recent compilers.

Compiler

Figure 2.24 - Hennessy & Patterson



Optimisation Classification

- High-level Optimisation
  - Performed on source and fed to later optimisations passes.
- Local Optimisation
  - Optimises code in a straight-line code fragment.
- Global Optimisation
  - Extends local optimisation across branches (loops).
- Register Allocation
  - Links registers with operands.
- Processor Dependent Optimisations
  - Take advantage of specific architectural knowledge.

Register Allocation

- Register allocation has a high impact on performance.
- Algorithms are based on graph coloring
  - Create a graph that provides candidates for register allocation
  - Limit set of colors so that no two adjacent nodes in a dependency graph have the same color.
  - Try to allocate registers for all active variables
  - Is NP-complete but heuristic algorithms run in near near-linear time (needs >=16 registers).

Optimisation Impact

- Difficult to separate local and processor-dependent optimisations from code-generator transformations.
- The following slide provides an example:
- The subsequent slide shows various optimisations
- The slide illustrates the importance of looking at optimised code.
  - May remove instructions completely

Optimisation Types

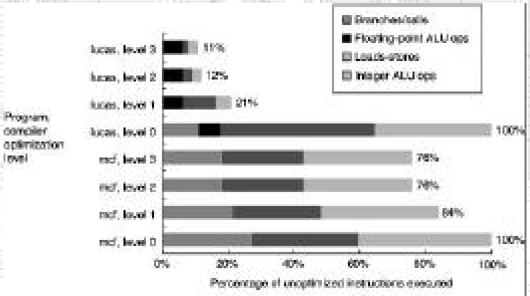
12 small FORTRAN and Pascal programs

Optimisation name	Explanation	Percentage of the total number of optimisation transformes
High-level	*At or near the source level:prosssor independent*	
Procedure integration	Replace procedure call by procedure body	N.M
Local	*Within straight-line code*	
Common subexpression elimination	Replace two instances of the same computation by single copy	18%
Constand propagation	Replace all instances of a variable that is assigned a constant with the constand	22%
Stack height reduction	Rearrange expression tree to minimize resources needed for expression evaluation	N.M
Global	*Across a branch*	
Global common subexpression elimination	Same as local, but this version crosses branches	13%
Copy propagation	Replace all instances of a variable A that has been assigned X(i.e.,A=X) with X	11%
Code motion	Remove code from a loop that computes same value each iteration of the loop	16%
Induction variable	Simplify/eliminate array addressing calculation within loops	2%
Processor dependent	*Depends on processor knowledge*	
Strenght reduction	Many example, such as replace multiply by a constant with add and shifts	N.M.
Pipeline scheduling	Reorder instruction to improve pipeline performace	N.M
Branch offset optimisation	Choose the shortes branch displacement that reaches target.	N.M

N.M. = Not Measured

Change in Instruction Counts (Optimisation level)

Figure 2.26 - Hennessy & Patterson



Impact of Compiler Technology Architecture

- Compiler and high-level languages affects the use of the ISA.
- Important questions:
  - How are variables allocated and addressed?
  - How many registers are needed to allocate variables appropriately?
- **High-level languages allocate data:**
  - On the stack to allocate local variables
  - In global data area to allocate statically declared objects
  - On the heap to allocate dynamic objects

Aliased Variables

- Register allocation for heap objects is impossible because they use pointers
- The same problem may arise for global and stack objects.

```
p  = &a     /* gets address of a in p */
a  = ...    /* assigns to a directly */
*p = ...    /* uses p to assign to a */
...a...     /* accesses a */
```