EEL 4768: Computer Architecture

Instruction Set Architecture (ISA)
(continued…)

Instructor: Zakhia (Zak) Abichar

Department of Electrical Engineering and Computer Science
University of Central Florida

"Computer Architecture, A Quantitative Approach", **Appendix A**
"Computer Organization and Design, The Hardware/Software Interface", **Section 2.15**

Encoding an Instruction Set

- The choices made in designing the Instruction Set Architecture (ISA) affect how the instructions are encoded and affect the implementation of the CPU

Instruction encoding

- If the ISA includes many addressing modes and instructions, the instruction size becomes larger since more bits are needed to distinguish the various options

The CPU

- The datapath looks at the instructions' fields and moves the data around to make the computations specified by the instructions
- A simple instruction encoding is beneficial for the hardware design

2

Encoding an Instruction Set

- Instruction encoding is usually done in two ways based on this question

How many possible addressing modes are there for an instruction? And how many operands does an instruction have?

Case 1:

- If there is a large number of operands and addressing modes, the instruction will have an **'address specifier'** and **'address'** field for every operand
- In this case, the opcode doesn't specify the addressing mode; the 'address specifier' does
- The opcode only specifies the operation, such as 'ADD', 'SUB', …

3

- Let's consider an architecture supports multiple addressing modes:

ADD <reg>, <reg>, <reg>
ADD <reg>, <reg>, <mem>
ADD <reg>, <mem>, <mem>
ADD <reg>, <reg>, <constant>
ADD <reg>, <mem>, <constant>
ADD <mem>, <mem>, <reg>

- The instruction format with 'address specifiers' can be used

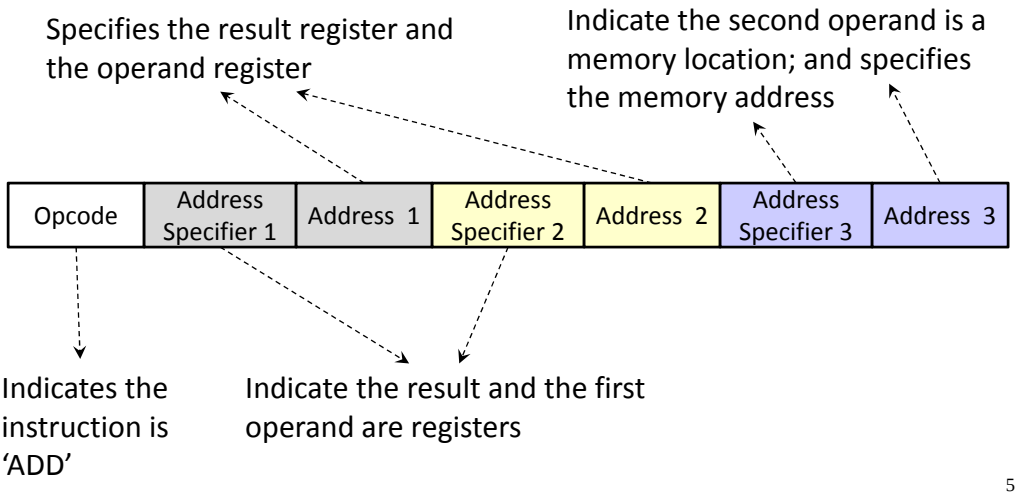
Opcode	Address Specifier 1	Address 1	Address Specifier 2	Address 2	Address Specifier 3	Address 3

- The 'address specifier' field indicates if an operand is a register, a memory location or a constant

4

- Example: let's consider this instruction

ADD <reg>, <reg>, <mem>



5

Case 2:

- An instruction could have a few addressing modes
- For example:

ADD <reg>, <reg>, <reg>

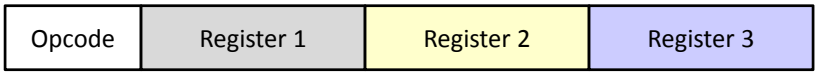
ADD <reg>, <reg>, <const>

- In this case, the addressing mode is encoded in the opcode
- There is no 'address specifier' field in the instruction

6

- This instruction can be encoded as:

ADD <reg>, <reg>, <reg>



- This instruction could use a different format:

ADD <reg>, <reg>, <const>

- For the two instructions above, the opcode indicates the format of the remaining of the instruction

7

Encoding an Instruction Set

- Another goal of instruction encoding is to have an instruction size that facilitates hardware design
- Nowadays, all the instructions are in multiple of bytes
  - 8-bit, 16-bit, 32-bit, 64-bit
  - Intel x86 allows an arbitrary number from 1 to 17 bytes (still a multiple of byte)

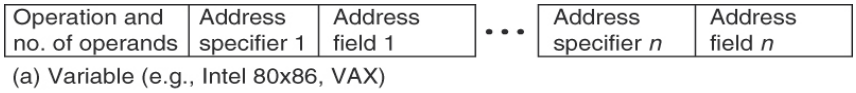
8

Encoding an Instruction Set

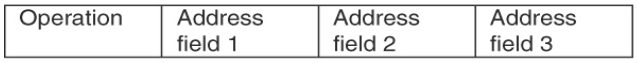
- The figure (next slide) shows the three main approaches of instruction encoding
- With the variable-length approach, the instruction length varies
  - Usually, it's always a multiple of 1 byte
- With the fixed-length approach, all the instruction take the same number of bits
- The hybrid approach specifies a few fixed-length instruction format; any instruction chooses one of these formats

9

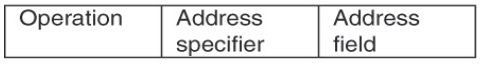
Encoding an Instruction Set

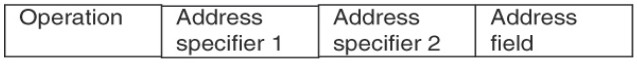


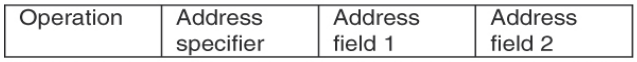
(a) Variable (e.g., Intel 80x86, VAX)



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







(c) Hybrid (e.g., IBM 360/370, MIPS16, Thumb, TI TMS320C54x)

10

Encoding an Instruction Set

- Architectures that support a lot of addressing modes usually use the variable-length encoding
- Let's say an architecture supports these modes:

<operation>	**<reg>, <reg>**
<operation>	**<reg>, <constant>**
<operation>	**<reg>, <memory>**
<operation>	**<memory>, <memory>**

- If the last instruction uses a direct memory address, we have:

ADD [100], [104] // add mem data 100 to mem data 104

- Let's say the opcode is 8-bit and the memory address is 32-bit
- This instruction would need (1+4+4) = 9 bytes to encode

(cont'd)

11

Encoding an Instruction Set

- However, the first format wouldn't need that many bytes
- The first format is: **operation <reg>, <reg>**
- Example: **ADD R1, R2**
- Let's also assume the opcode is 8-bit
- If there are 32 registers, then the register address is 5-bit
- The instruction length needs to be: (8+5+5) = 18 bits, which can be done with 3-byte instruction

- Therefore, this architecture needs 3-byte and 9-byte instructions
- If we make it fixed-length, every instruction should be 9-byte
- This means the 3-byte instruction will waste 6 bytes

- Obviously, such an architecture should be encoded with variable-length instructions

12

Encoding an Instruction Set

- Architectures that have a few addressing modes for an instruction are suitable for the fixed-length approach
- For example, an instruction might allow only register and immediate operands for arithmetic instructions:

  ADD <reg>, <reg>, <reg>

  ADD <reg>, <reg>, <constant>

- Let's assume the opcode is 8-bit and there are 32 registers (a register need 5-bit field)
- The first format above needs (8+5+5+5) = 23 bits; can be done with 3-byte instruction
- The second format can be done with (8+5+5+14) = 32 bits; can be done with 4-byte instruction (the constant field is 14 bits)

(cont'd)

13

Encoding an Instruction Set

- Let's assume these are the load and store instructions:

  LOAD <reg>, <reg>, <constant>

  STORE <reg>, <reg>, <constant>

- There's only one addressing mode for loads & stores: <reg>+<constant>
- These two instructions can be encoded with (8+5+5+14) = 32 bits
- They can be done with 4-byte instruction

Conclusion:

- The 'ADD' instruction needs 3 or 4 bytes
- The 'LOAD' and 'STORE' need 4 bytes
- Therefore, let's use a fixed instruction length of 4 bytes
- It's true the ADD instruction with 3 bytes would waste some space, but the hardware to fetch instructions will be simple since the next instruction is always the next 4 bytes from the memory

14

Encoding an Instruction Set

Conclusion from the two previous examples…

- Numerous addressing modes lead to some instructions being large (those that have memory addresses)
- Then, it's not practical anymore to use the largest instruction size for all the instructions
- Therefore, a variable-instruction length is used

- Few addressing modes makes it possible to use a reasonably small number of bits for all the instructions
- The instructions more or less use the same number of bits
- It becomes reasonable to use a fixed-length instruction format

15

Encoding an Instruction Set

Which approach (fixed-length or variable-length) reduces the size of the code?

•In general, the variable-length reduces the size of the code

•This is the example we've seen earlier:

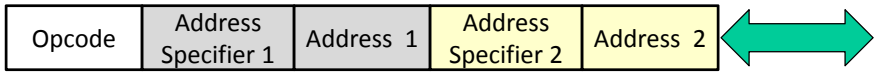
ADD	**<reg>, <reg>, <reg>**	**3 bytes**
ADD	**<reg>, <reg>, <constant>**	**4 bytes**
LOAD	**<reg>, <reg>, <constant>**	**4 bytes**
STORE	**<reg>, <reg>, <constant>**	**4 bytes**

•By using 4-byte fixed instruction, the first ADD instruction is actually wasting 1 byte

16

Encoding an Instruction Set

- On the other hand, the variable-length instruction doesn't waste bits in the instruction since the instruction size is adaptable



The size of the instruction is adaptable:

- If the instruction needs more fields, it will include them
- If not, no bits will be wasted

17

Encoding an Instruction Set

Which approach (fixed-length or variable-length) reduces the size of the code?

•Another aspect of this question is that usually variable-length instructions allow multiple addressing modes for arithmetic

•For example:

```
ADD    R1, Mem
```

•The instruction above can add an operand from the memory without having to load first

•Whereas, in the fixed-length instruction the add might only allow register operands

•The equivalent code would be:

```
LOAD   R2, Mem
ADD    R1, R2
```

•Therefore, the fixed-length approach required using more instructions which increases the code size

18

Reduced Code Size in RISC

- The full-featured RISC architectures usually use 32-bit instructions
  - A 16-bit instruction is not sufficient since it's not possible to have a large number of registers, operations and a decent size immediate field
- One problem with 32-bit instructions in embedded applications is the large code size
  - For the same number of instructions, a 32-bit code takes twice the amount of memory, in comparison to 16-bit code
  - And it consumes more battery power to read the code
- There are multiple solutions to address this problem…

19

Reduced Code Size in RISC

- One way to reduce the code size for embedded applications is to use a simpler variant of the architecture that doesn't support all the features
  - MIPS came up with the MIPS16 architecture for embedded devices
  - ARM came up with the Thumb architecture
- The simpler architecture has these features:
  - The instruction is smaller (16-bit vs 32-bit)
  - The architecture doesn't use all the registers (eg: can only use 16 registers, therefore, the register field uses 4 bits)
  - The instruction uses two operands instead of three operands (eg: 'ADD R1, R2' instead of 'ADD R1, R2, R3')
  - The number of instructions is smaller (the opcode becomes smaller)
  - The immediate field is smaller

20

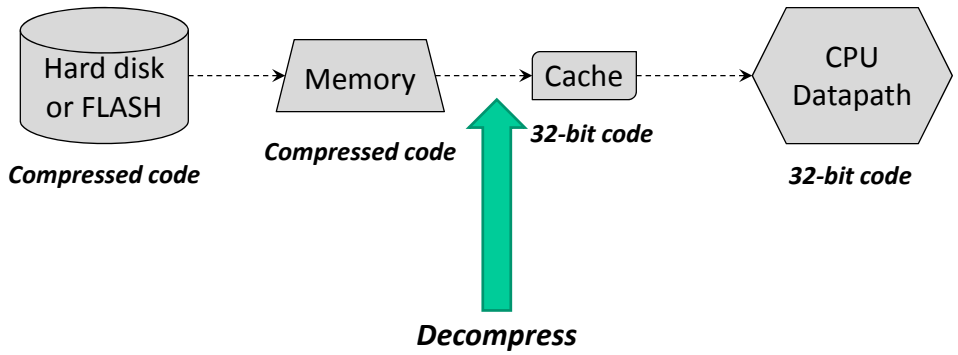
Reduced Code Size in RISC

- Another approach to reduce the code size in embedded devices is to keep the code 32-bit but to compress it
  - This approach is like compressing a file into a .zip file that becomes smaller
- IBM computers used such an approach in a technology called 'CodePack' in the PowerPC processors
  - The instruction is stored as compressed code
  - It's decompressed in the instruction cache right before it goes into the datapath
  - The datapath only sees the 32-bit code

21

Reduced Code Size in RISC

- IBM's CodePack approach
  - The storage (hard disk or FLASH) size is reduced
  - The memory size is reduced
  - The datapath is based on 32-bit instructions



22

Reduced Code Size in RISC

Let's compare the two approaches (inventing a simpler instruction vs. compressing the code)…

- Inventing a simpler architecture (eg: MIPS 16, Thumb)
  - This basically brings a new architecture to the picture
  - We need a new compiler
  - We also need new hardware to run the new type of instructions
  - **Performance:** The code size is reduced by up to 40%

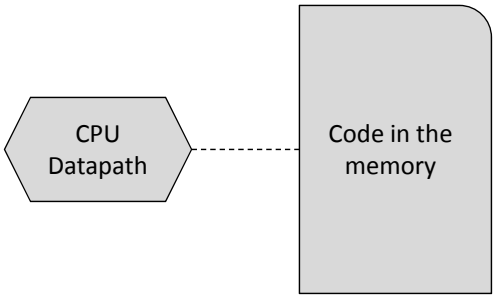
- Compressing the instructions (eg: CodePack)
  - The compiler and hardware stay the same (based on 32-bit code)
  - **Performance:** The code size is reduced 35% to 40% due to compression
  - **Performance:** The code runs a bit slower since it should be uncompressed before it goes in the datapath

23

Reduced Code Size in RISC

- The compression technology also brings additional overhead
- Let's say the code in the CPU branches to address 4000
- The code is compressed in the memory, which means the target address is not at address 4000 anymore

- A hash table is maintained which maps 'uncompressed addresses' to 'compressed addresses'
- The table tells us where instruction address 4000 is located in the memory



24

The Role of Compilers

The Role of Compilers

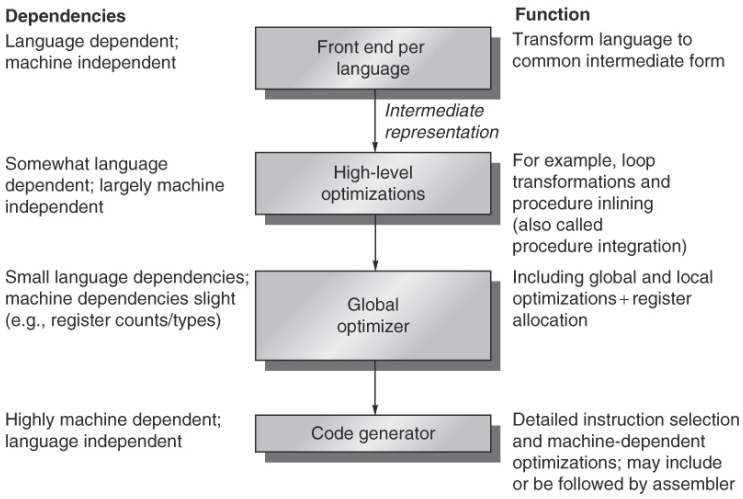
- Today, almost all programming for desktop and server applications is done in high-level languages
  - Therefore, we rely on the compiler to generate the machine code that runs on the processor
- In embedded applications, assembly programming is still used in some cases
  - This allows the programmer to reduce the size of the code to the smallest possible
  - This is done since embedded systems might have a very small memory

The Role of Compilers

- Earlier architectures included multiple features in the assembly language since programming was done in assembly
- Nowadays, in most cases, the machine code is generated by the compiler
  - The assembly language (and architecture) should work well with the compiler
  - The goal is to have the generated code fast, small in size and correct

27

The Role of Compilers

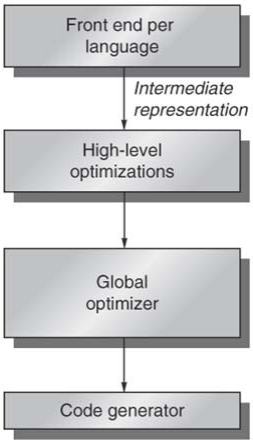


- This is the structure of a compiler
- The code is transformed in multiple phases from a high-level code into a machine code

28

- The first phase is language-dependent (since it takes the high-level code)
- The subsequent phases change the code to low-level languages
- Finally, the code generator comes up with the machine code

- Sometimes, we can reuse the lower phases for multiple languages (C, C++)
- This means compilers of two different languages (eg C, C++) can share all the lower components; they would have different components for the first level (Front end per language)



29

The Role of Compilers

- The 'front end' part of the compiler takes the high-level language and transforms it into an **'Intermediate Representation'**

- The 'intermediate representation' looks like an assembly language; it's made of simple instructions
- The compiler then applies multiple optimizations on the 'intermediate representation'
- Finally, the optimized 'intermediate representation' is converted to machine code

30

The Role of Compilers

- Let's consider this piece of code:

save: an array

i: index in the array

k: a variable (containing a constant)

- This piece of code loops over the array as long as the element is equal to 'k'
- It stops at the first position that's different from 'k'

```
while (save[i] == k)
    i += 1;
```

31

The Role of Compilers

(1) Scanning

- First, the front end will scan the code and create a string of **'tokens'**
- A token is a single word in the code that's either a reserved name, a variable name or an operation
- Basically, the scanning separates the multiple words in the code

```
while (save[i] == k)
    i += 1;
```

String of tokens:
while, (, save, [, i,], ==, k,), i, +=, 1

32

The Role of Compilers

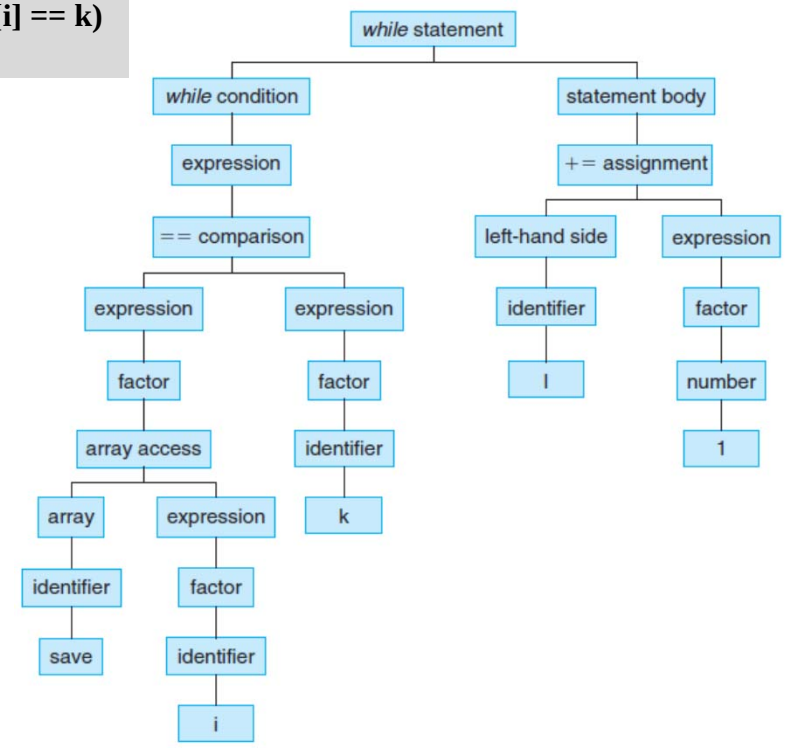
(2) Parsing

- Parsing takes the token string and ensures that the syntax is correct
- The parsing phase generates an **'abstract syntax tree'** based on the code (next slide)

String of tokens:
while, (, save, [, i,], ==, k,), i, +=, 1

33

while (save[i] == k)
i += 1;



34

The Role of Compilers

(3) Semantic analysis

- The semantic analysis takes the abstract syntax tree and checks the semantics of the language
- It checks that the variables and types are properly declared (called **'type checking')**
- It also checks that the operators and objects match

- A **'symbol table'** is built which contains the names of all the variables that are used in the code

35

The Role of Compilers

(4) Generation of the intermediate representation

- The 'symbol table' and the 'abstract syntax tree' are used to generate the intermediate representation (next slide)
- This is the output of the 'front end'

- The intermediate representation looks like generic assembly language
- However, it assumes that there is an infinite number of registers (virtual registers) since it doesn't run on the hardware
- The intermediate representation is used by the other parts of the compiler which apply the optimizations

- Once the code is represented through the intermediate representation, the remaining parts of the compiler are independent on the high-level language in which the code is written

36

- This is the intermediate representation of the code

```
while (save[i] == k)
  i += 1;
```

```
loop:
LI      R1, save        # loads the starting address of save into R1
LW      R2, i
MULT R3, R2, 4          #Multiply R2 by 4 assuming 32-bit data
ADD     R4, R3, R1
LW      R5, 0(R4)       # load save[i]
LW      R6, k
BNE     R5, R6, endwhileloop

LW      R6, i
ADD     R7, R6, 1       # increment
SW      R7, i
branch loop             # next iteration
endwhileloop:
```

37

The Role of Compilers

- If you look at the code in the previous slide, you can realize that it's not the shortest the code possible
- It has been generated through the abstract syntax tree…
- The next steps of the compiler, the optimizations, will enhance the performance of the code

38

The Role of Compilers

- This is another code in 'intermediate representation' that has been generated (before it was optimized)
- This code is quite inefficient; it computes the address of x[i] twice even though it's the same address

x[i] = x[i] + 4

```
LI    R100, x                 # address of array
LW    R101, i                 # value of i
MULT  R102, R101, 4           # data is 32-bit
ADD   R103, R100, R102
LW    R104, 0(R103)
ADD   R105, R104, 4           # increment the data by 4
LI    R106, x
LW    R107, i                 # compute the address of x[i]
MULT  R108, R107, 4           # again!
ADD   R109, R106, R107
SW    R105, 0(R109)
```

39

The Role of Compilers

- This is one transformation of the code after some optimizations are applied
- The new code below recognizes that the variables, x[i], are the same and computes the array address once in R103

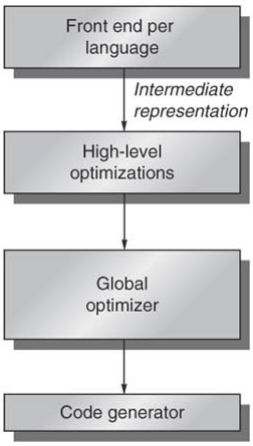
Notice the use of register R100; the intermediate representation uses any register number; later, they will be mapped to the actual hardware registers

x[i] = x[i] + 4

```
LI    R100, x                 # load address of the array
LW    R101, i                 # load the variable 'i'
MULT  R102, R101, 4
ADD   R103, R100, R102
LW    R104, 0(R103)           # load the data
ADD   R105, R104, 4           # increment the data by 4
SW    R105, 0(R103)           # store the data
```

40

- The compiler in the figure is called an **'optimizing compiler'**



- The compiler makes multiple optimizations on the code
- The optimizations are usually optional
- Doing more optimizations may reduce the number of instructions in the generated code; thus, the code will run faster
- However, doing more optimizations will increase the compilation time

- The optimizations can be done in various order sometimes
- That is, optimization 1 and optimization 2 can be applied in any order
- Determining the order is called the **'phase ordering problem'**

41

The Role of Compilers

- The table (next slide) shows some of the common optimizations in the compiler
- The number in the right side of the table shows how frequently an optimization is used
- A compiler could use some optimizations, not all of them
  - While more advanced compilers may use all of these optimizations
  - The numbers in the table show the popular optimizations
  - The term **'N.M.'** means the optimization was 'Not Measured'

42

Optimization name	Explanation	Percentage of the total number of optimizing transforms
High-level	*At or near the source level; processor-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%
Constant propagation	Replace all instances of a variable that is assigned a constant with the constant	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 elimination	Simplify/eliminate array addressing calculations within loops	2%
Processor-dependent	*Depends on processor knowledge*	
Strength reduction	Many examples, such as replace multiply by a constant with adds and shifts	N.M.
Pipeline scheduling	Reorder instructions to improve pipeline performance	N.M.
Branch offset optimization	Choose the shortest branch displacement that reaches target	N.M.

43

The Role of Compilers

- A **'pass'** made by the compiler over the code is also called a **'phase'**

These are the common optimizations:

- **'Global common subexpression elimination'**
  - An expression that shows up in the code multiple times is computed only once; the result is saved in a temporary variable; for subsequent uses the temporary variable is used
  - This reduces the computations since the result used frequently is computed once only
  - The disadvantage is that a new variable is created to hold the temporary result

44

The Role of Compilers

'Global common subexpression elimination'

- In this code, the term (pi*r^2) is computed twice
- Therefore, this term is a 'common subexpression'

```
a1 = (pi*r*r) *d;          // on term is a circle's area: (pi*r*r)
a2 = (pi*r*r) * (e+f);     // the same term is used here
```

- The code is transformed as below, by computing (pi*r*r) once and storing it in a temporary variable

```
temp = pi * r * r;
a1 = temp *d;              // on term is a circle's area: (pi*r*r)
a2 = temp * (e+f);         // the same term is used here
```

- The optimized code uses less computations

45

The Role of Compilers

- **Constant propagation**
  - If a variable is equal to a constant, we can replace the occurrences of that variable by the constant (eliminating the variable)
  - Eg: x=3; y=x+a; z=x*b;
  - It becomes: y=a+3; z=b*3; (variable 'x' is removed)
- **Copy propagation**
  - After a statement such as 'A=X', all the occurrences of A are replaced with X
  - Then, there are fewer variables to track and allocate to registers

46

The Role of Compilers

- **Code motion**
  - This optimization removes a code from a loop that computes the same value in each iteration of the loop
  - Eg: while(save[i] == k) i+=1;
  - If we load 'k' in the loop, it will load the same value in every iteration
  - We should load 'k' outside of the loop

47

The Role of Compilers

- **Induction variable elimination**
  - This optimization simplifies or eliminates array addressing calculations within loops
  - Example: let's access the variables in an array
  - The array start address is in register 's0'
  - We want to loop over the elements A[0] up to A[9]
  - The addresses of the elements are: s0, s0+4, …, s0+36

48

The Role of Compilers

- This code changes 't0' from 0 to 9
- In the loop, it multiplies 't0' by 4, then it adds this value to 's0' to obtain the address of the elements

Computing the addresses (100, 104, …, 136)

```
        addi    t0, zero, zero      // initialize t0 to zero
        addi    t7, zero, 10        // condition to stop the loop
Loop:   sll     t1, t0, 2           // shift left by 2 bits (multiply by 4)
        add     t1, t1, s0          // this is the address of an element
        …
        addi    t0, t0, 1
        bne     t0, t7, loop        // if t0 isn't yet equal to 10, loop one more time
```

- *The next slide shows how the address computation can be improved*

49

The Role of Compilers

- The register 't0' is initialized to 's0', which is the first address
- We copied 's0' into 't0' since we'll modify t0 (the address in 's0' will be unchanged)
- Then, we can increment 't0' by 4 every iteration to obtain the next address

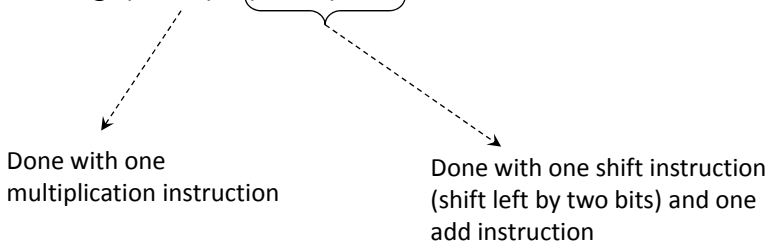
Computing the addresses (100, 104, …, 136)

```
        add     t0, s0, zero        // initialize t0 to s0, address of the first element
        addi    t7, s0, 40          // stop address
Loop:   …                           // when the loop starts, the address is t0
        addi    t0, t0, 4
        bne     t0, t7, loop
```

50

The Role of Compilers

- **Strength reduction**
  - This phase replaces multiplications by constants with shifts instructions and adds
  - Sometimes doing a shift operation and multiple add operations takes fewer clock cycles than doing one multiplication instruction
  - Eg: (t0 * 5) = (t0 * 4) + t0



51

The Role of Compilers

- The compiler makes multiple passes on the code
- In each pass, an optimization is done without knowing what the subsequent passes will do
  - This is a challenge for compilers
  - The optimization done in a pass can be useless (or even slow the code) if the next pass takes a contradictory action
- The compiler doesn't look at the actions of all the passes jointly
  - This is too complex to do
  - It also makes the compilation time too large

  (cont'd)

52

The Role of Compilers

- One example is the 'global common subexpression elimination'
- The code below was optimized by introducing a new variable, temp, equal to (pi*r*r)
  - This result was computed once and used multiple times

```
a1 = (pi*r*r) *d;              // on term is a circle's area: (pi*r*r)
a2 = (pi*r*r) * (e+f);         // the same term is used here
```

53

The Role of Compilers

- The downside of the optimized code (below) is using one extra variable, the temporary variable

```
temp = pi * r * r;
a1 = temp *d;                  // on term is a circle's area: (pi*r*r)
…                              // multiple lines of code
a2 = temp * (e+f);             // the same term is used here
```

- At the time we're evaluating 'a2', if 'temp' is in a register, then the optimized code runs faster since it uses less computations
- However, there might be multiple lines of code between 'a1' and 'a2' and that might push 'temp' out of the registers and into the cache
- Therefore, we should do a 'load' before we can use 'temp'
- Or worse, 'temp' might be pushed out of the cache and into the RAM which takes 100s of cycles and the code might be slower than the original non-optimized version

54

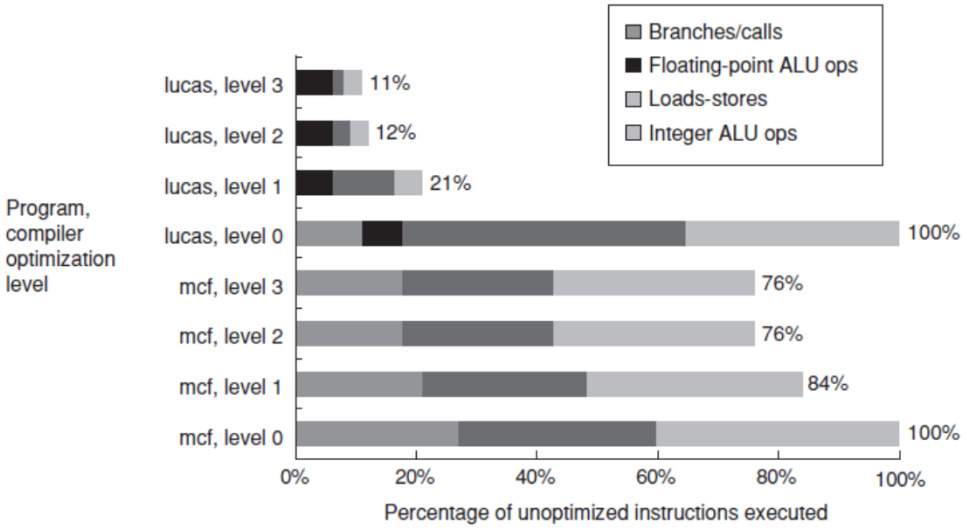
The Role of Compilers

- *What could the architecture designer do about this situation?*
- The architecture designer could include multiple general-purpose registers (32 instead of 16) to reduce the chance that the 'temp' doesn't find a register

55

The Role of Compilers

- The figure shows how the optimizations reduces the number of instructions executed



56

The Role of Compilers

- The figure shows the results of two benchmark programs 'lucas' and 'mcf'
- Level 0 refers to the non-optimized code (this is the code in intermediate representation which is usually quite inefficient)
- Higher levels indicate that more optimizations have been applied
- The figure shows how the compiler optimizations reduce the number of instructions executed in the code
- The 'lucas' program got a very large drop in the number of instructions executed
- The 'mcf' program still got a decent drop in the number of instructions

57

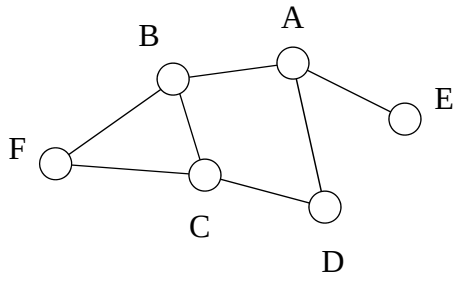
The Role of Compilers

Item #1: The architecture should provide at least 16 registers

- The variables in a program are mapped into a graph like below
- Two variables that are connected by a line means they are dependent; they also need to be in registers at the same time
- Allocating variables to register is equivalent to the **'graph coloring'** problem where each circle is colored such as each two adjacent circles have different colors (basically, each color is a register)

For example, we can see from the graph that A and B will have different colors; meaning they will be placed in difference registers since they're related and they need to be in registers at the same time.

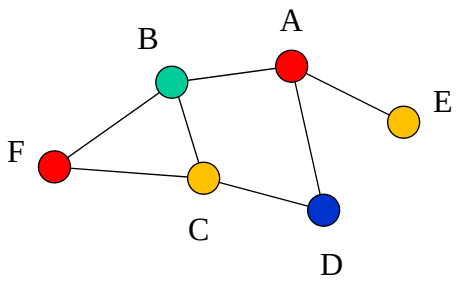
How can we color this graph?



58

The Role of Compilers

- This is one possible way to color the graph
- We used four colors; therefore, we're using 4 registers to allocate the variables



59

The Role of Compilers

- The graph coloring problem is NP-complete; this means it takes exponential time to solve (can't be done in polynomial time)
- Instead of using an optimal solution, the compiler uses a **'heuristic algorithm'** which is an approximation algorithm that colors the graph

- The question is: *how good is the heuristic algorithm?*
- If it's close to optimal, then the register allocation is good

- It turns out, when there are few registers, the heuristic algorithm has a bad performance
- However, with more registers (16 or more) the heuristic algorithm is close to optimal
- **Therefore, having 16 or more registers in the architecture simplifies the compiler's task of allocating register**

60

The Role of Compilers

- Another challenge with register allocation is the use of pointers
- In the code below the variable 'a' can be accessed either through its name or through the pointer 'ptr'
- If 'a' is kept in a register, a 'store word' instruction might be applied on the pointer; changing 'a' in the memory and not in the register
- Therefore, 'a' in the register is not the most updated value

```
int a;
int *ptr = &a;    // ptr is pointing to a

a = …             // a is mapped to a register
*ptr = …          // a is changed in the memory but not in the register!
y = a + …         // a is used from the register; not the latest value!
```

- In such a case, the compiler would avoid storing 'a' in a register
- It would load it, do some operations and immediately store it back in the memory

61

The Role of Compilers

Item #2: The assembly instructions should provide regularity to simplify the compiler's task

- Regularity means there should be orthogonality between the main aspects: (1) addressing modes, (2) data types, and (3) operations
- Let's say the addressing modes are: register and immediate
- The data types are: signed integer and unsigned integer
- The operations are: add and sub
- *(this is just an example, in fact there are about 5 addressing modes, 7 data types and tens or hundreds of instructions)*
- Orthogonality means that each operation should support all the data types and all the addressing modes

62

The Role of Compilers

- With orthogonality, the instructions below should be supported

```
Add     Reg, Reg, Reg          # signed
Addi    Reg, Reg, Immediate
Sub     Reg, Reg, Reg
Subi    Reg, Reg, Immediate
Addu    Reg, Reg, Reg          # unsinged
Addui   Reg, Reg, Immediate
Subu    Reg, Reg, Reg
Subui   Reg, Reg, Immediate
```

What's the advantage of orthogonality?

- The compilers will have many instructions to choose from, which makes it easier to generate the code

63

The Role of Compilers

Does MIPS provide orthogonality?

- Yes, it attempts to do so
- We have: add, addi, and, andi, or, ori

- However, we don't have 'subi' (subtract immediate)
- Why? Because its function can be done with 'addi'

- Rather than doing: subi $t0, $t0, 4
- We can do addi $t0, $t0, -4

- *Conclusion:* providing orthogonality simplifies the compiler's task; but the instruction set doesn't need to have redundant instructions

64

The Role of Compilers

- Another way to interpret orthogonality is that any register should be used with any instruction to provide flexibility for the compiler

Does MIPS provide orthogonality based on register use?

- Mostly, yes; for example, the R-type can use any register ($t0-$t7, $s0-$s7 and the others)
- However, 'jal' (jump-and-link) doesn't provide orthogonality since it always links in register $ra
- If 'jal' can link in any register, that would be better from the compiler's point of view

- The VAX architecture's 'BSR' (Branch to SubRoutine) instruction provides orthogonality on the register use

BSR R1, Label # return address goes in any reg. (R1 in this case)

65

The Role of Compilers

- **<u>Item #3:</u> The architecture should provide primitives (simple instructions) not solutions (complex instructions)**
- Some architectures try to provide instructions that match high-level languages to make the compiler's job easier
- However, such attempts are often not successful

- The complex instructions are often specialized that they may not match exactly to what the compiler is trying to generate
- Therefore, the compiler won't utilize these instructions and will use multiple simple instructions instead

66

The Role of Compilers

- One such example is the 'CALLS' (Call Subroutine) instruction in the VAX architecture
- The 'CALLS' instruction tries to take care of all the details of calling a procedure, which are the tasks below:

1) Push the number of arguments on the stack
2) Push the argument themselves on the stack
3) Save the registers that will be altered by the procedure
4) Push the return address on the stack
5) Clear the condition codes
6) Update the two stack pointers (top and bottom)
7) Branch to the first instruction of the procedure

67

The Role of Compilers

- The downside of 'CALLS' is that it's an overkill; it does more computations than is actually needed
- For example, most procedures already know the number of arguments they take; there's no need to pass the number of arguments on the stack
- Some procedures may not need to use the conditions codes; therefore, clearing the condition codes wastes the CPU cycles
- …
- The conclusion is, the architecture should provide simple instructions (building blocks) that can be used to build any functionality the compiler is trying to generate, efficiently

68

The Role of Compilers

- This is another example of a complex instruction
- The assembly language provides one instruction to do a loop that's similar to the for-loop in a high-level language

C code: for(a=0; a<=100; a++) { … }

Assembly: loop R1, 100 … endloop

- So the assembly instruction above initializes R1 to 0 and iterates until R1 becomes 100, incrementing R1 by 1 in each iteration
- The problem with the assembly instruction is that it might not map to all the ways of doing a loop

69

The Role of Compilers

- The high-level code might want to decrement the counter
- Or, the high-level code might want to increment the counter by 2
- To support all such features, the 'loop' instruction would have to be provided in multiple modes which would take multiple opcodes to distinguish and uses transistors to implement
- Secondly, if the assembly provides 10 variants of the 'loop' instruction, maybe 3 of them will be used for 80% of loops
- Therefore, the other 7 modes are supported but they're only used with a small probability
- Therefore, we could say the other 7 modes shouldn't be supported and they shouldn't take up transistors on the chip for their implementation

70

The Role of Compilers

<u>Item #4:</u> Help the compiler writer to determining which code sequence is faster

- The compiler usually targets the architecture which is defined as 'the low-level software environment'
- However, the architecture can be implemented in various ways called the **'microarchitecture'** (for example, a MIPS CPU can be single-cycle, multi-cycle or pipelined)
- The compiler produces an executable that runs on any implementation of the architecture
- However, the challenge is that the various implementations may each favor a particular code generation by the compiler

71

The Role of Compilers

Example

- Let's consider the strength reduction optimization
- The multiplication was replaced by a shift and add

Regular Code	Optimized Code
mult $t0, $t0, 5 #multiply by 5	sll $t1, $t0, 2 #shift left 2 is like multiply by 4 add $t0, $t1, $t0

- These are two possible implementations of the same CPU

Implementation #1	Implementation #2
Mult: 10 cycles (using short cycles) Shift: 3 cycles Add: 3 cycles	Mult: 1 cycle (using long cycles) Shift: 1 cycle Add: 1 cycle

- In the first implementation, the optimized code is better (6 vs 10)
- In the second implementation, the optimized code is worse (2 vs 1)

72

The Role of Compilers

- *Based on the previous example, how can the compiler determine which code is better? … given that the compiler is not aware of the implementation*

- Well, the compiler doesn't know…
- However, the architecture inventors may try to give some hints as to what instructions are usually faster
- For example, the architecture inventors may claim that 'shift-adds' are highly likely to be faster than multiplications; therefore, the compiler will use the optimization

73

The Role of Compilers

<u>Item #5:</u> An assembly instruction should not compute a value that's known at compile time

- The CALLS subroutine uses a few clock cycles to compute the number of parameters for the subroutine
- However, this number is known at compile time
- A good compiler would compute the number of parameters during compilation

74

The Role of Compilers

- The conclusion…
- For an architecture to work well with the compiler… the architecture inventors should:
- Increase the number of registers (16 or more)
- Use simple instructions
- Clarify what instructions are likely to be faster
- Computations that can be done at compile time should not be done at runtime

In the past: the compiler has nothing to do with the architecture… they were considered as two separate layers

Now: the architecture is the compiler target… they should interact well to generate a code that is fast and efficient

75