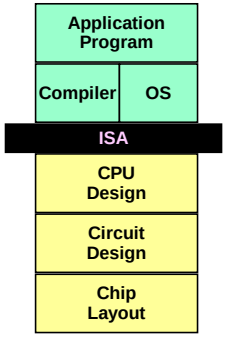
Instruction Set Architecture

- Assembly Language View
  - Processor state
    - Registers, memory, …
  - Instructions
    - `addl`, `movl`, `leal`, …
    - How instructions are encoded as bytes
- Layer of Abstraction
  - Above: how to program machine
    - Processor executes instructions in a sequence
  - Below: what needs to be built
    - Use variety of tricks to make it run fast
    - E.g., execute multiple instructions simultaneously



Basic Issues in Instruction Set Design

Goal: find a language that makes it easy to build both the hardware and the compiler while maximizing performance and minimizing cost

- What operations (and how many) should be provided
- How (and how many) operands are specified
- What data types and sizes
- How to encode these into consistent instruction formats

Typical Operations

- Data movement
- Arithmetic
- Logical
- Shift
- Control (jump/branch)
- Subroutine linkage
- Interrupt
- Synchronization
- Multimedia

Types of ISAs/instructions

- Stack architectures
  - All operations refer to operands on stack and implicitly operate on the top of stack
- Accumulator architectures
  - Implicitly access single register + other explicitly named operands
- General register architectures
  - Operands can be found in either registers or memory
- Load-store architectures
  - All operands found in registers; only load/store operations access memory

Addressing Modes

- Register
- Immediate
- Register indirect
- Displacement
- Indexed
- Direct (or absolute)
- Memory indirect
- Auto-increment
- Scaled indexed

175

Scaled Indexed Addressing Mode

- Most General Form

  D(Rb,Ri,S) Mem[Reg[Rb]+S*Reg[Ri]+ D]

  - D: Constant "displacement" 1, 2, or 4 bytes
  - Rb: Base register: Any of 8 integer registers
  - Ri: Index register: Any, except for `%esp`
  - S: Scale: 1, 2, 4, or 8
- Special Cases

(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]]
D(Rb,Ri)	Mem[Reg[Rb]+Reg[Ri]+D]
(Rb,Ri,S)	Mem[Reg[Rb]+S*Reg[Ri]]

176

Instruction Format

- Can be
  - fixed length – simpler decoding
  - variable length – higher code density/smaller program
  - Large instruction word (encoding multiple instructions and dependences among those instructions)
- If we have many memory operands per instruction and many addressing modes, we need an address specifier per operand/result
- If we have a load-store architecture with 1 address per instruction and one or two addressing modes, we can encode the addressing mode in the opcode

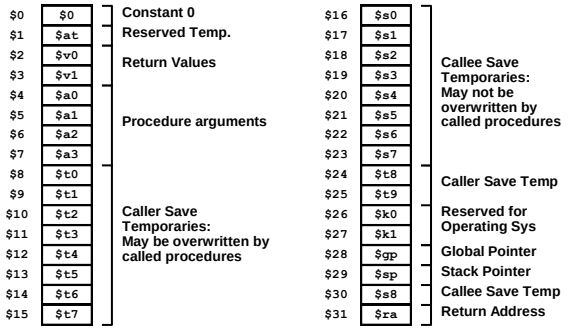
177

RISC Instruction Sets

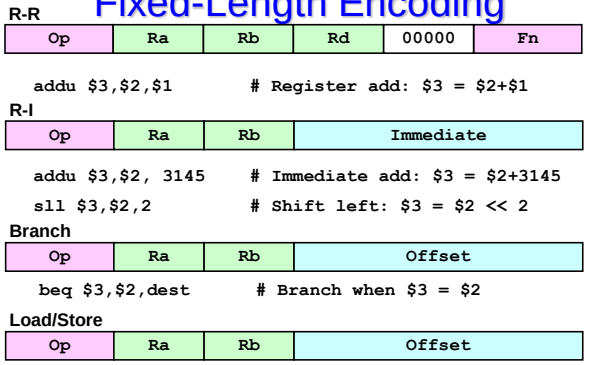
  - Reduced Instruction Set Computer
  - Internal project at IBM, later popularized by Hennessy (Stanford) and Patterson (Berkeley)
- Fewer, simpler instructions
  - Might take more to get given task done
  - Can execute them with small and fast hardware
- Fixed length encoding
- Simple addressing formats (typically just base+displacement)
- Register-oriented instruction set
  - Many more (typically 32) registers
  - Use for arguments, return pointer, temporaries
- Only load and store instructions can access memory
- No Condition codes - test instructions return 0/1 in register
- Implementation artifacts exposed to machine-level programs

178

Example: MIPS Registers



MIPS Instruction Examples: Fixed-Length Encoding



CISC Instruction Sets

- Complex Instruction Set Computer
- Dominant style through mid-80's

- Philosophy
  - Add instructions to perform "typical" programming tasks
- Variable length encodings
- Variable execution times
- Multiple formats for specifying operands
- Implementation artifacts hidden from machine-level programs
- Stack-oriented instruction set
  - Use stack to pass arguments, save program counter
  - Explicit push and pop instructions
- Any instruction can access memory
  - `addl %eax, 12(%ebx,%ecx,4)`
    - requires memory read and write
    - Complex address calculation
- Condition codes
  - Set as side effect of arithmetic and logical instructions

CISC vs. RISC

- Original Debate
  - Strong opinions!
  - CISC proponents---easy for compiler, fewer code bytes
  - RISC proponents---better for optimizing compilers, can make run fast with simple chip design
- Current Status
  - For desktop processors, choice of ISA not a technical issue
    - With enough hardware, can make anything run fast
    - Code compatibility more important
  - For embedded processors, RISC makes sense
    - Smaller, cheaper, less power

183

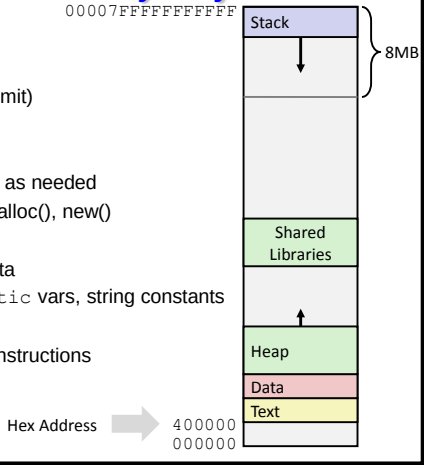
Data Layout in Memory

184

x86-64 Linux Memory Layout

not drawn to scale

- Stack
  - Runtime stack (8MB limit)
  - E. g., local variables
- Heap
  - Dynamically allocated as needed
  - When call malloc(), calloc(), new()
- Data
  - Statically allocated data
  - E.g., global vars, `static` vars, string constants
- Text / Shared Libraries
  - Executable machine instructions
  - Read-only

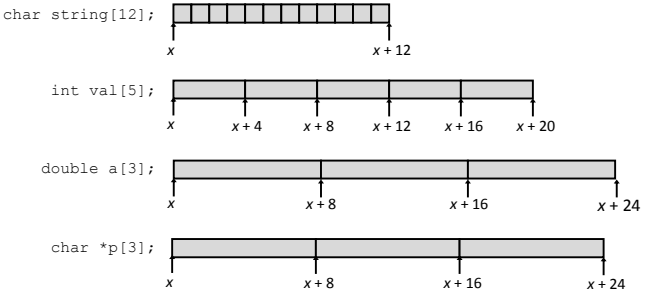


185

Array Allocation

- Basic Principle

  T **`A[`***L***`];`**
  - Array of data type *T* and length *L*
  - Contiguously allocated region of *L* * **`sizeof`**`(`*T*`)` bytes in memory



186

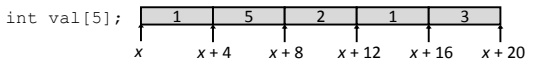
Array Access

- Basic Principle

  T **A[***L***];**

  – Array of data type *T* and length *L*

  – Identifier **A** can be used as a pointer to array element 0: Type *T**



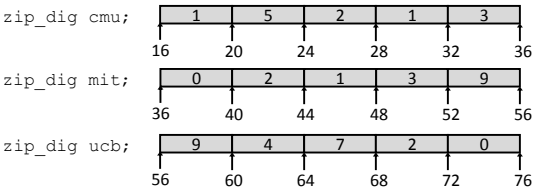
- Reference

Reference	Type	Value
`val[4]`	`int`	3
`val`	`int *`	*x*
`val+1`	`int *`	*x* + 4
`&val[2]`	`int *`	*x* + 8
`val[5]`	`int`	??
`*(val+1)`	`int`	5
`val + i`	`int *`	*x* + 4*i*

187

Array Example

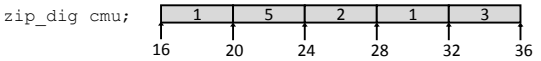
```
#define ZLEN 5
typedef int zip_dig[ZLEN];

zip_dig cmu = { 1, 5, 2, 1, 3 };
zip_dig mit = { 0, 2, 1, 3, 9 };
zip_dig ucb = { 9, 4, 7, 2, 0 };
```



- Declaration "`zip_dig cmu`" equivalent to "`int cmu[5]`"
- Example arrays were allocated in successive 20 byte blocks
  - Not guaranteed to happen in general

188

Array Accessing Example




```
int get_digit
  (zip_dig z, int digit)
{
  return z[digit];
}
```

IA32

```
  # %rdi = z
  # %rsi = digit
movl (%rdi,%rsi,4), %eax  # z[digit]
```

- Register `%rdi` contains starting address of array
- Register `%rsi` contains array index
- Desired digit at `%rdi + 4*%rsi`
- Use memory reference `(%rdi,%rsi,4)`

189

Array Loop Example

```
void zincr(zip_dig z) {
  size_t i;
  for (i = 0; i < ZLEN; i++)
    z[i]++;
}
```

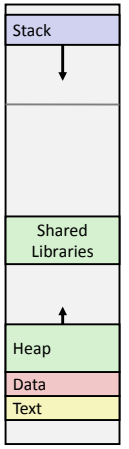
```
  # %rdi = z
  movl    $0, %eax          #   i = 0
  jmp     .L3               #   goto middle
.L4:                        # loop:
  addl    $1, (%rdi,%rax,4) #   z[i]++
  addq    $1, %rax          #   i++
.L3:                        # middle
  cmpq    $4, %rax          #   i:4
  jbe     .L4               #   if <=, goto loop
  rep; ret
```

190

Memory Allocation Example

not drawn to scale

```
char big_array[1L<<24];  /* 16 MB */
char huge_array[1L<<31]; /*  2 GB */

int global = 0;

int useless() { return 0; }

int main ()
{
    void *p1, *p2, *p3, *p4;
    int local = 0;
    p1 = malloc(1L << 28); /* 256 MB */
    p2 = malloc(1L << 8);  /* 256  B */
    p3 = malloc(1L << 32); /*   4 GB */
    p4 = malloc(1L << 8);  /* 256  B */
 /* Some print statements ... */
}
```



Where does everything go?

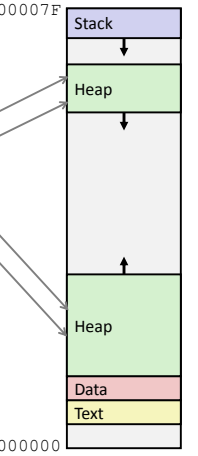
191

x86-64 Example Addresses

not drawn to scale

address range ~2^{47}

local	0x00007ffe4d3be87c
p1	0x00007f7262a1e010
p3	0x00007f7162a1d010
p4	0x000000008359d120
p2	0x000000008359d010
big_array	0x0000000080601060
huge_array	0x0000000000601060
main()	0x000000000040060c
useless()	0x0000000000400590



192

Breakout

- Pointers in C

  long i, *pi, **ppi;

  i = 4; pi = &i; ppi = & pi;

Assume i is located at 0x8000; pi is located at 0x8008; and ppi is located at 0x8010

What value is contained in *a* for the following statements –

a = ppi+1;

a = &ppi;

*a = **ppi;*

*a = *(pi+1);*

*a = *(*ppi+1);*

193

193

Today

- Arrays
  - One-dimensional
  - Multi-dimensional (nested)
  - Multi-level
- Structures
  - Allocation
  - Access
  - Alignment
- Floating Point

194

Multidimensional (Nested) Arrays

- Declaration

  T **A**[*R*][*C*];

  – 2D array of data type *T*

  – *R* rows, *C* columns

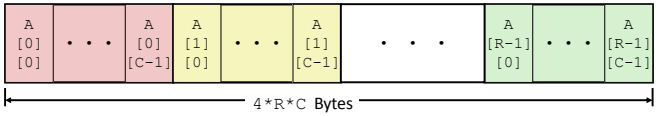
  – Type *T* element requires *K* bytes

- Array Size

  – *R* * *C* * *K* bytes

```
A[0][0]   . . .   A[0][C-1]
   .                  .
   .                  .
   .                  .
A[R-1][0] . . .  A[R-1][C-1]
```

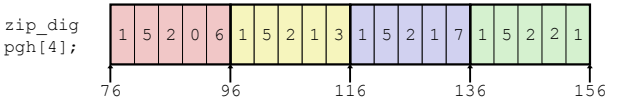
```
int A[R][C];
```



195

Nested Array Example

```
#define PCOUNT 4
zip_dig pgh[PCOUNT] =
  {{1, 5, 2, 0, 6},
   {1, 5, 2, 1, 3 },
   {1, 5, 2, 1, 7 },
   {1, 5, 2, 2, 1 }};
```



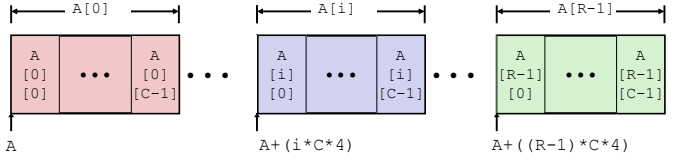
- "`zip_dig pgh[4]`" equivalent to "`int pgh[4][5]`"
  - Variable **`pgh`**: array of 4 elements, allocated contiguously

196

Nested Array Row Access

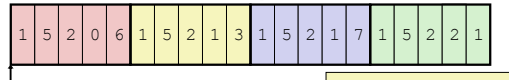
- Row Vectors
  - **`A[i]`** is array of *C* elements
  - Each element of type *T* requires *K* bytes
  - Starting address **`A`** + *i* * (*C* * *K*)

```
int A[R][C];
```



197

Nested Array Row Access Code




```
int *get_pgh_zip(int index)
{
  return pgh[index];
}
```

```
# %rdi = index
 leaq (%rdi,%rdi,4),%rax # 5 * index
 leaq pgh(,%rax,4),%rax  # pgh + (20 * index)
```

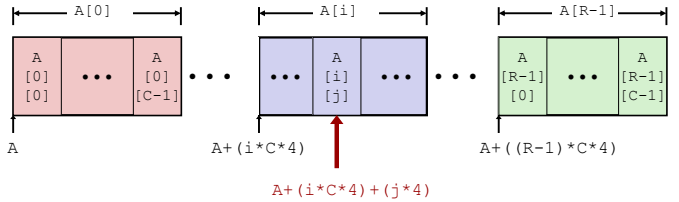
- Row Vector
  - **`pgh[index]`** is array of 5 **`int`**'s
  - Starting address **`pgh+20*index`**
- Machine Code
  - Computes and returns address
  - Compute as **`pgh + 4*(index+4*index)`**

198

Nested Array Element Access

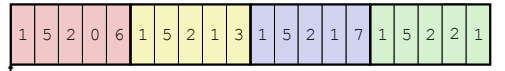
- Array Elements
  - **`A[i][j]`** is element of type *T,* which requires *K* bytes
  - Address **`A`** + *i* * (*C* * *K*) + *j* * *K* = *A* + (*i* * *C* + *j*)* *K*

```
int A[R][C];
```



199

Nested Array Element Access Code




```
int get_pgh_digit
   (int index, int dig)
{
  return pgh[index][dig];
}
```

```
leaq   (%rdi,%rdi,4), %rax    # 5*index
addl   %rax, %rsi             # 5*index+dig
movl   pgh(,%rsi,4), %eax     # M[pgh + 4*(5*index+dig)]
```

- Array Elements
  - **`pgh[index][dig]`** is **`int`**
  - Address: **`pgh + 20*index + 4*dig`**
    - = **`pgh + 4*(5*index + dig)`**

200