Chapter 2

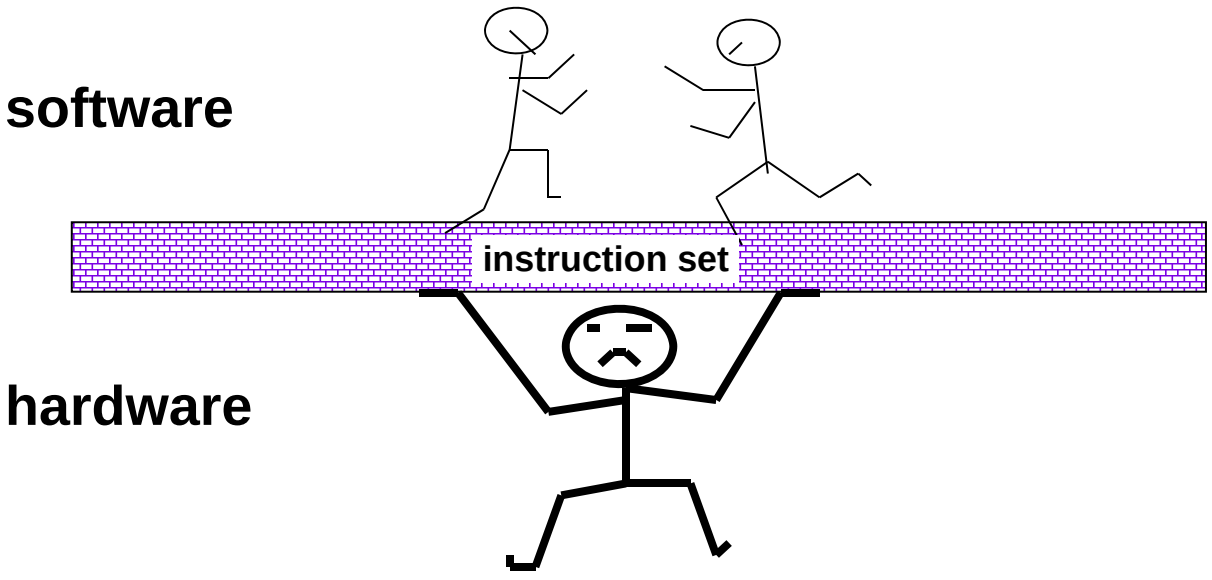
Instructions: Language of the Computer

Tien-Fu Chen

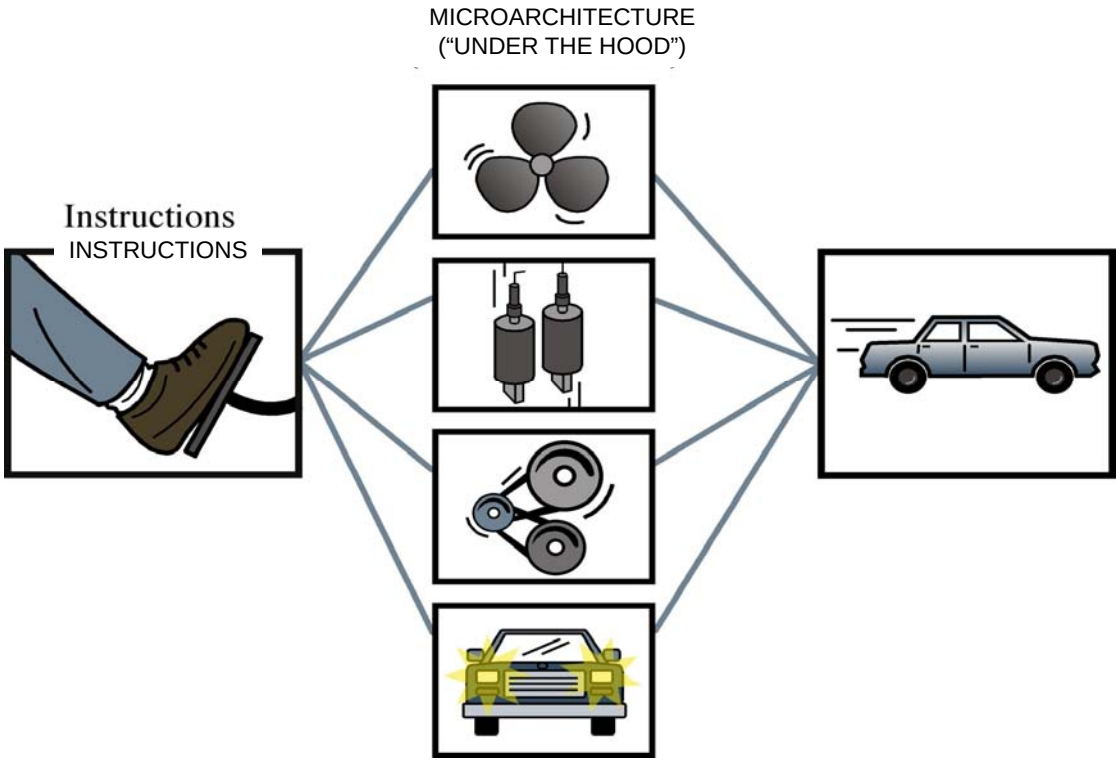
Dept. of Computer Science and Information Engineering

National Chiao Tung Univ.

Instruction Set Design



Instructions are Like Car Pedals



See Patt & Patel text, Andelman Decl. in support of ARM's Motion for Summary Judgment, Ex. C, at 12.

Instruction Sets

- Instruction Set

  An agreement between architects and machine language programmers
- Aspects
  - Operations
    - Arithmetic and logical +, -, X, /...
    - Data Transfer - load/store
    - Control - branch, jump, call, return
    - Floating Point operations - FADD, FDIV
    - String
    - System support
    - HLL (high-level languages)
  - Operands
    - operand storage
    - number of operands
    - addressing operands
    - Type and size of operands
    - Implicit/Explicit operands

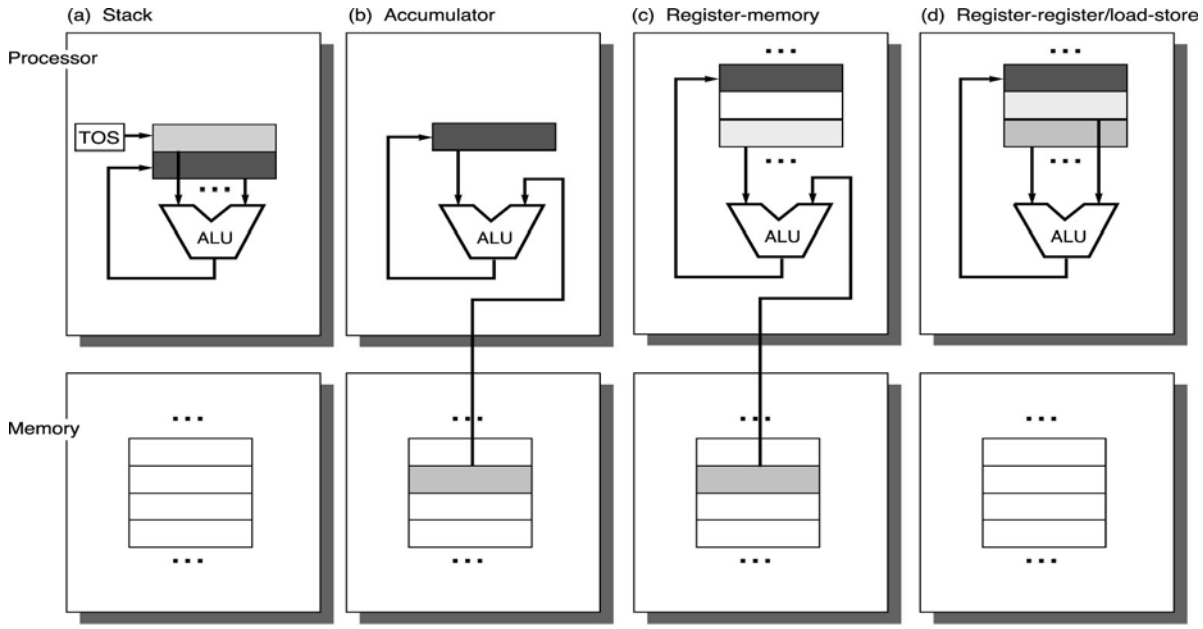
3 Classes of Instruction operations

- Data movement instructions
  - Move data from a memory location or register to another memory location or register without changing its form
  - *Load*—source is memory and destination is register
  - *Store*—source is register and destination is memory
- Arithmetic and logic (ALU) instructions
  - Change the form of one or more operands to produce a result stored in another location
  - *Add*, *Sub*, *Shift*, etc.
- Branch instructions (control flow instructions)
  - Alter the normal flow of control from executing the next instruction in sequence
  - *Br Loc*, *Brz Loc2*,—unconditional or conditional branches

Classifying ISA by location of operands

- **Stack**: 0 address
  add (sp)<- (sp) + (sp-1)
- **Accumulator**: 1 address, 1+x address
  add A acc=acc + mem[A]
  addx A acc=acc + mem[A + x]
- **Memory-memory**
  add A,B EA(A)=EA(A) + EA(B)
  add A B C EA(A)=EA(B) + EA(C)
- **Register-memory** via general purpose register: **2,3 address**
  add R1, A R1 <- R1 + mem[A]
- **Load/Store machine**: 3 address
  add Ra Rb Rc Ra=Rb + Rc
  load Ra Rb Ra= mem[Rb]
  store Ra Rb mem[Rb]= Ra
  - **access memory only with load and store instructions**

Classifying ISAs



Comparing Number of Instructions

Code sequence for (C = A + B) for four classes of instruction sets:

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

Issues of Operands

- **Endian Convention**

  Ordering the bytes within a word

  - **Big Endian**: MSB at xx00

word addr	MSB			LSB
0	0	1	2	3
4	4	5	6	7

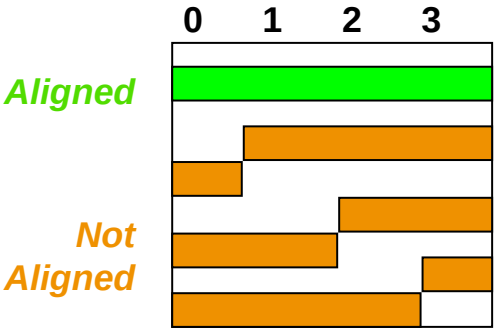
ex. : IBM, Motorola

  - **Little Endian**: LSB at xx00

word addr	MSB			LSB
0	3	2	1	0
4	7	6	5	4

ex. Intel, Dec

- **Aligned vs. Misaligned**



ISA Design Principles

- *Design Principle 1:* Simplicity favors regularity
  - Regularity makes implementation simpler
  - Simplicity enables higher performance at lower cost
- *Design Principle 2:* Smaller is faster
  - c.f. main memory: millions of locations
- *Design Principle 3:* Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction
- *Design Principle 4:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

RISC-V Instruction Set

The RISC-V Instruction Set

- Used as the example throughout the book
- Developed at UC Berkeley as open ISA
- Now managed by the RISC-V Foundation (riscv.org)
- Typical of many modern ISAs
  - See RISC-V Reference Data tear-out card
- Similar ISAs have a large share of embedded core market
  - Applications in consumer electronics, network/storage equipment, cameras, printers, …

Register Operands

- Arithmetic instructions use register operands
- RISC-V has a 32 × 64-bit register file
  - Use for frequently accessed data
  - 64-bit data is called a “doubleword”
    - 32 x 64-bit general purpose registers x0 to x30
  - 32-bit data is called a “word”
- ***Design Principle 2:*** Smaller is faster
  - c.f. main memory: millions of locations

RISC-V Registers

Name	Register number	Usage	Preserved on call?
x0	0	The constant value 0	n.a.
x1 (ra)	1	Return address (link register)	yes
x2 (sp)	2	Stack pointer	yes
x3 (gp)	3	Global pointer	yes
x4 (tp)	4	Thread pointer	yes
x5-x7	5–7	Temporaries	no
x8-x9	8–9	Saved	yes
x10-x17	10–17	Arguments/results	no
x18-x27	18–27	Saved	yes
x28-x31	28–31	Temporaries	no

x0: the constant value 0
x1: return address
x2: stack pointer
x3: global pointer
x4: thread pointer
x5 – x7, x28 – x31: temporaries
x8: frame pointer
x9, x18 – x27: saved registers
x10 – x11: function arguments/results
x12 – x17: function arguments

Register Operand Example

- C code:

```
f = (g + h) - (i + j);
```

  - f, …, j in x19, x20, …, x23

- Compiled RISC-V code:

```
add x5, x20, x21
add x6, x22, x23
sub x19, x5, x6
```


Memory Operands

- Main memory used for composite data
  - Arrays, structures, dynamic data
- To apply arithmetic operations
  - Load values from memory into registers
  - Store result from register to memory
- Memory is byte addressed
  - Each address identifies an 8-bit byte
- RISC-V is Little Endian
  - Least-significant byte at least address of a word
  - *c.f.* Big Endian: most-significant byte at least address
- RISC-V does not require words to be aligned in memory

Memory Operand Example

- C code:

```
A[12] = h + A[8];
```

  - h in x21, base address of A in x22

- Compiled RISC-V code:
  - Index 8 requires offset of 64
    - 8 bytes per doubleword

```
ld      x9, 64(x22)
add     x9, x21, x9
sd      x9, 96(x22)
```


Registers vs. Memory

- Registers are faster to access than memory
- Operating on memory data requires loads and stores
  - More instructions to be executed
- Compiler must use registers for variables as much as possible
  - Only spill to memory for less frequently used variables
  - Register optimization is important!

Immediate Operands

- Constant data specified in an instruction

```
addi x22, x22, 4
```

- ***Design Principle 3:*** Make the common case fast
  - Small constants are common
  - Immediate operand avoids a load instruction

Sign Extension

- Representing a number using more bits
  - Preserve the numeric value
- Replicate the sign bit to the left
  - c.f. unsigned values: extend with 0s
- Examples: 8-bit to 16-bit
  - +2: 0000 0010 => 0000 0000 0000 0010
  - –2: 1111 1110 => 1111 1111 1111 1110
- In RISC-V instruction set
  - `lb`: sign-extend loaded byte
  - `lbu`: zero-extend loaded byte

Representing Instructions

- Instructions are encoded in binary
  - Called machine code
- RISC-V instructions
  - Encoded as 32-bit instruction words
  - Small number of formats encoding operation code (opcode), register numbers, …
  - Regularity!

32-bit RISC-V Instruction Formats

Instruction Formats	31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16	15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
Register/register	funct7							rs2					rs1					funct3			rd					opcode						
Immediate	imm[11:0]												rs1					funct3			rd					opcode						
Upper Immediate	imm[31:12]																				rd					opcode						
Store	imm[11:5]							rs2					rs1					funct3			imm[4:0]					opcode						
Branch	[12]	imm[10:5]						rs2					rs1					funct3			imm[4:1]				[11]	opcode						
Jump	[20]	imm[10:1]										[11]	imm[19:12]								rd					opcode						

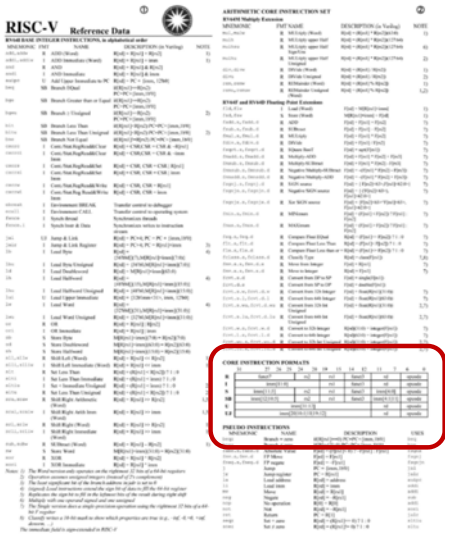
- **opcode (7 bit):** partially specifies which of the 6 types of *instruction formats*
- **funct7 + funct3 (10 bit):** combined with **opcode**, these two fields describe what operation to perform
- **rs1 (5 bit):** specifies register containing first operand
- **rs2 (5 bit):** specifies second register operand
- **rd (5 bit):**: Destination register specifies register which will receive result of computation

RISC-V Instruction Format

RISC-V Encoding Summary

Name (Field Size)	Field 7 bits	5 bits	5 bits	3 bits	5 bits	7 bits	Comments
R-type	funct7	rs2	rs1	funct3	rd	opcode	Arithmetic instruction format
I-type	immediate[11:0]		rs1	funct3	rd	opcode	Loads & immediate arithmetic
S-type	immed[11:5]	rs2	rs1	funct3	immed[4:0]	opcode	Stores
SB-type	immed[12,10:5]	rs2	rs1	funct3	immed[4:1,11]	opcode	Conditional branch format
UJ-type	immediate[20,10:1,11,19:12]				rd	opcode	Unconditional jump format
U-type	immediate[31:12]				rd	opcode	Upper immediate format

- R-Format: instructions using 3 register inputs
  - add, xor, mul —arithmetic/logical ops
- I-Format: instructions with immediates, loads
  - addi, lw, jalr, slli
- S-Format: store instructions: sw, sb
- SB-Format: branch instructions: beq, bge
- U-Format: instructions with upper immediates
  - lui, auipc —upper immediate is 20-bits
- UJ-Format: jump instructions: jal


RISC-V Reference Data

R-format Instructions

funct7	rs2	rs1	funct3	rd	opcode
7 bits	5 bits	5 bits	3 bits	5 bits	7 bits

- Instruction fields
  - opcode: operation code
  - rd: destination register number
  - funct3: 3-bit function code (additional opcode)
  - rs1: the first source register number
  - rs2: the second source register number
  - funct7: 7-bit function code (additional opcode)

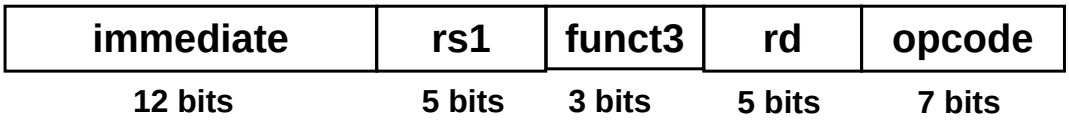
R-format Example


```
add x9,x20,x21
```

0	21	20	0	9	51
0000000	10101	10100	000	01001	0110011

$0000\ 0001\ 0101\ 1010\ 0000\ 0100\ 1011\ 0011_{two} = 015A04B3_{16}$

I-format Instructions

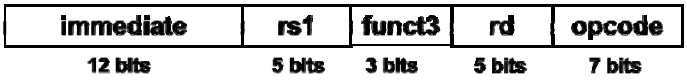


- Immediate arithmetic and load instructions
  - rs1: source or base address register number
  - immediate: constant operand, or offset added to base address
    - 2s-complement, sign extended
- ***Design Principle 3**:* Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

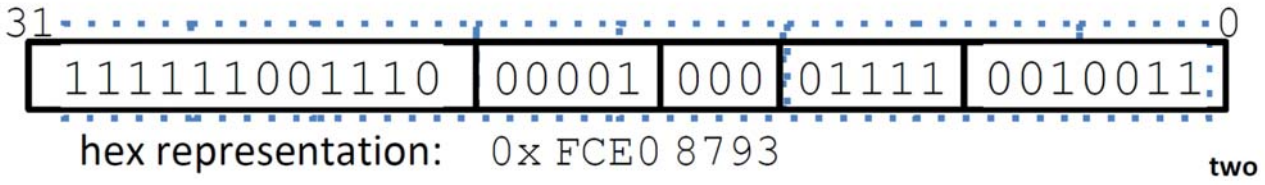
I-Format Example

- I-type Instruction:

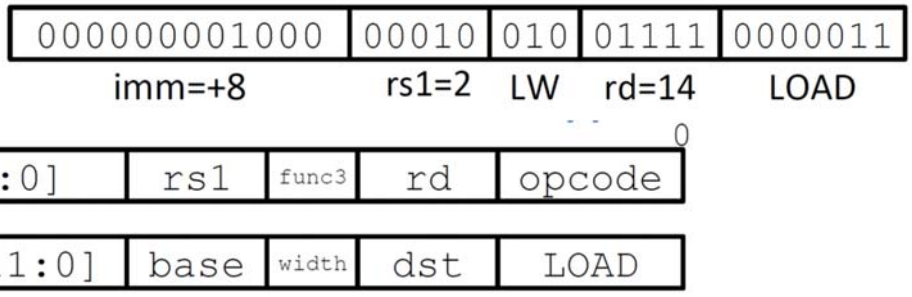
  addi x15,x1,-50



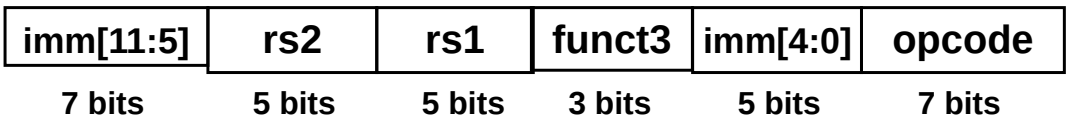
Field representation (binary):



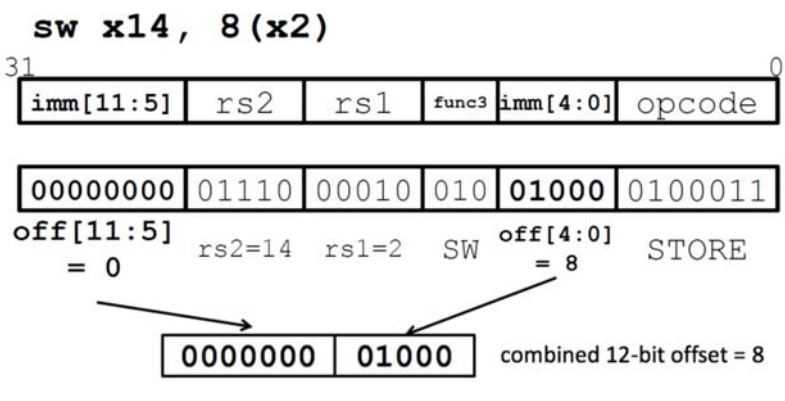
- **Load Instructions are also I-Type**
  - **lw x14, 8(x2)**



S-format Instructions



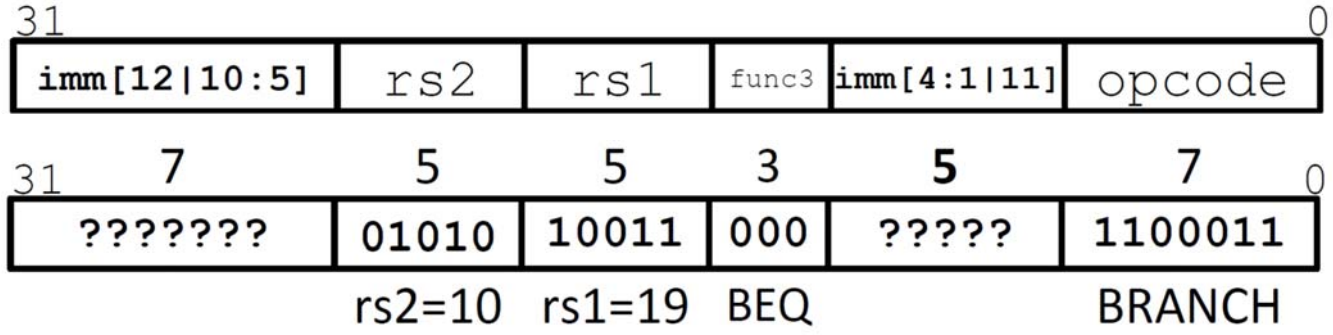
- Different immediate format for store instructions
  - rs1: base address register number
  - rs2: source operand register number
  - immediate: offset added to base address
    - Split so that rs1 and rs2 fields always in the same place



B-Format for Branches

- B-format is mostly same as S-Format, with two register sources (rs1/rs2) and a 12-bit immediate
- But now immediate represents values -2^{12} to $+2^{12}-2$ in 2-byte increments

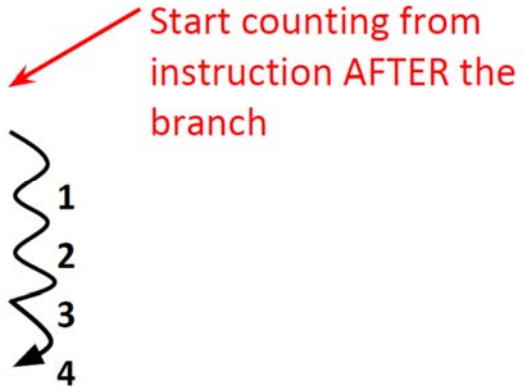
beq x19,x10,End



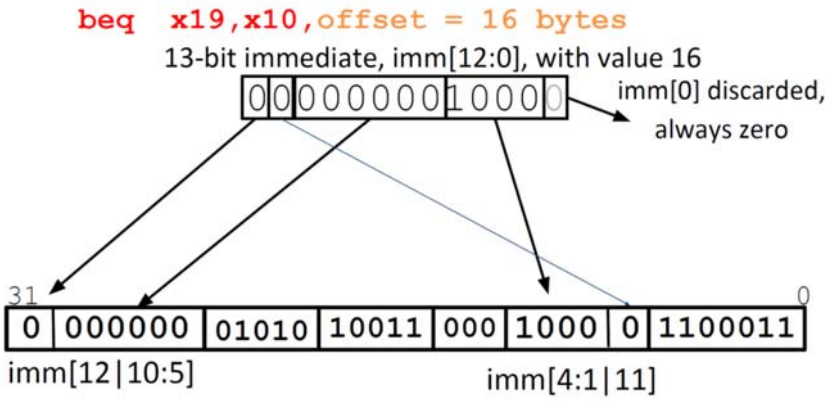
Branch Example

- RISCV Code:

```
Loop: beq  x19,x10,End
      add  x18,x18,x10
      addi x19,x19,-1
      j    Loop
End:  <target instr>
```

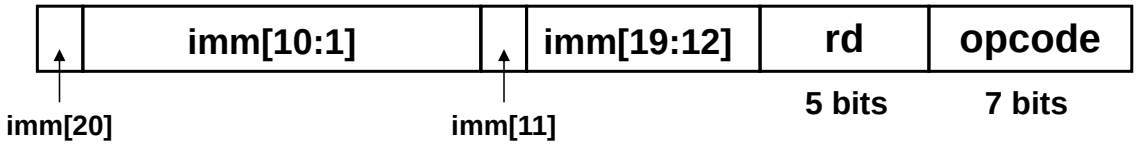


target = PC + sign-extend(imm13)
if GPR[rs1]==GPR[rs2]
- then PC <- target
- else PC <- PC + 4

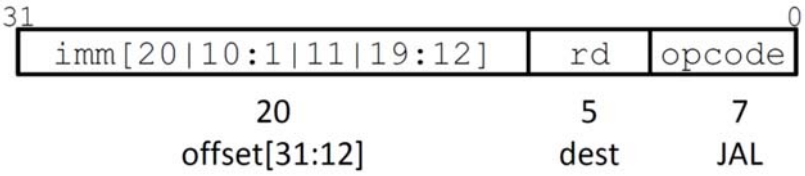


JU-Format for Jump

- Jump and link (`jal`) target uses 20-bit immediate for larger range



- jal saves PC+4 in register rd (the return address)
  - Set PC = PC + offset (PC-relative jump)
  - Target somewhere within $\pm 2^{19}$ locations, 2 bytes apart = $\pm 2^{18}$ 32-bit instructions
- "j" jump is a pseudo-instruction—the assembler will instead use jal but sets rd=x0 to discard return address
- Immediate encoding optimized similarly to branch instruction to reduce hardware cost



RISC-V Instruction Operations

Logical Operations

- Instructions for bitwise manipulation

Operation	C	Java	RISC-V
Shift left	<<	<<	`slli`
Shift right	>>	>>>	`srli`
Bit-by-bit AND	&	&	`and, andi`
Bit-by-bit OR	|	|	`or, ori`
Bit-by-bit XOR	^	^	`xor, xori`
Bit-by-bit NOT	~	~	

- Useful for extracting and inserting groups of bits in a word

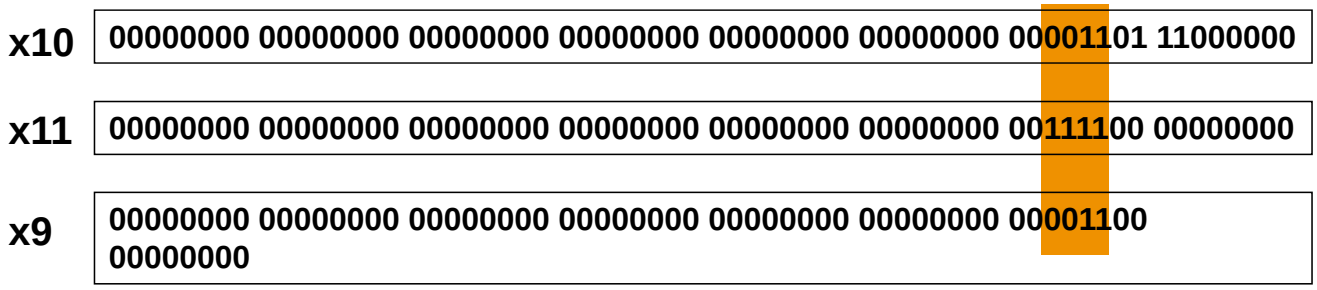
Shift Operations

- immed: how many positions to shift
- Shift left logical
  - Shift left and fill with 0 bits
  - `slli` by *i* bits multiplies by 2^i
- Shift right logical
  - Shift right and fill with 0 bits
  - `srli` by *i* bits divides by 2^i (unsigned only)

AND Operations

- Useful to mask bits in a word
  - Select some bits, clear others to 0

```
and x9,x10,x11
```



Conditional Operations

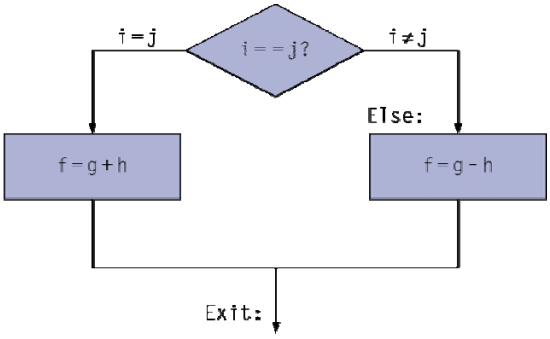
- Branch to a labeled instruction if a condition is true
  - Otherwise, continue sequentially
- `beq rs1, rs2, L1`
  - if (rs1 == rs2) branch to instruction labeled L1
- `bne rs1, rs2, L1`
  - if (rs1 != rs2) branch to instruction labeled L1

Compiling If Statements

- C code:

```
if (i==j) f = g+h;
else f = g-h;
```

  - f, g, … in x19, x20, …



- Compiled RISC-V code:

```
        bne x22, x23, Else
        add x19, x20, x21
        beq x0,x0,Exit // unconditional
Else:   sub x19, x20, x21
Exit:   …
```

Assembler calculates addresses

Compiling Loop Statements

- C code:

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

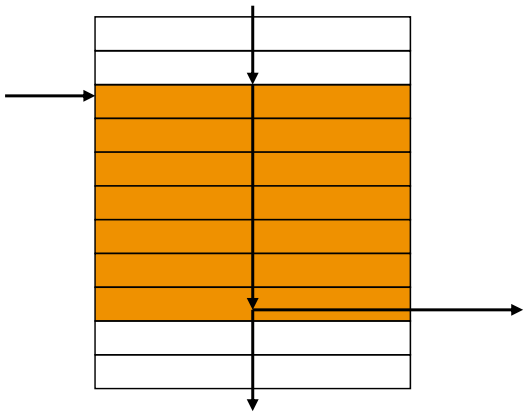
  - i in x22, k in x24, address of save in x25

- Compiled RISC-V code:

```
Loop: slli x10, x22, 3
      add  x10, x10, x25
      ld   x9, 0(x10)
      bne  x9, x24, Exit
      addi x22, x22, 1
      beq  x0, x0, Loop
Exit: …
```

Basic Blocks

- A basic block is a sequence of instructions with
  - No embedded branches (except at end)
  - No branch targets (except at beginning)



- **A compiler identifies basic blocks for optimization**
- **An advanced processor can accelerate execution of basic blocks**

More Conditional Operations

- `blt rs1, rs2, L1`
  - if (rs1 < rs2) branch to instruction labeled L1
- `bge rs1, rs2, L1`
  - if (rs1 >= rs2) branch to instruction labeled L1
- Example
  - if (a > b) a += 1;
  - a in x22, b in x23

```
    bge  x23, x22, Exit      // branch if b >= a
    addi x22, x22, 1
Exit:
```

Procedure Calling

- Steps required
  1. Place parameters in registers x10 to x17
  2. Transfer control to procedure
  3. Acquire storage for procedure
  4. Perform procedure's operations
  5. Place result in register for caller
  6. Return to place of call (address in x1)

Procedure Call Instructions

- Procedure call: jump and link

```
jal x1, ProcedureLabel
```

  - Address of following instruction put in x1
  - Jumps to target address

- Procedure return: jump and link register

```
jalr x0, 0(x1)
```

  - Like jal, but jumps to 0 + address in x1
  - Use x0 as rd (x0 cannot be changed)
  - Can also be used for computed jumps
    - e.g., for case/switch statements

Leaf Procedure Example

- C code:

```
long long int leaf_example (
   long long int g, long long int h,
   long long int i, long long int j) {
  long long int f;
  f = (g + h) - (i + j);
  return f;
}
```

  - Arguments g, …, j in x10, …, x13
  - f in x20
  - temporaries x5, x6
  - Need to save x5, x6, x20 on stack

Leaf Procedure Example

- RISC-V code:

```
leaf_example:
  addi sp,sp,-24
  sd   x5,16(sp)
  sd   x6,8(sp)
  sd   x20,0(sp
  add  x5,x10,x11
  add  x6,x12,x1
  sub  x20,x5,x6
  addi x10,x20,0
  ld   x20,0(sp)
  ld   x6,8(sp)
  ld   x5,16(sp)
  addi sp,sp,24
  jalr x0,0(x1)
```

Save x5, x6, x20 on stack

x5 = g + h

x6 = i + j

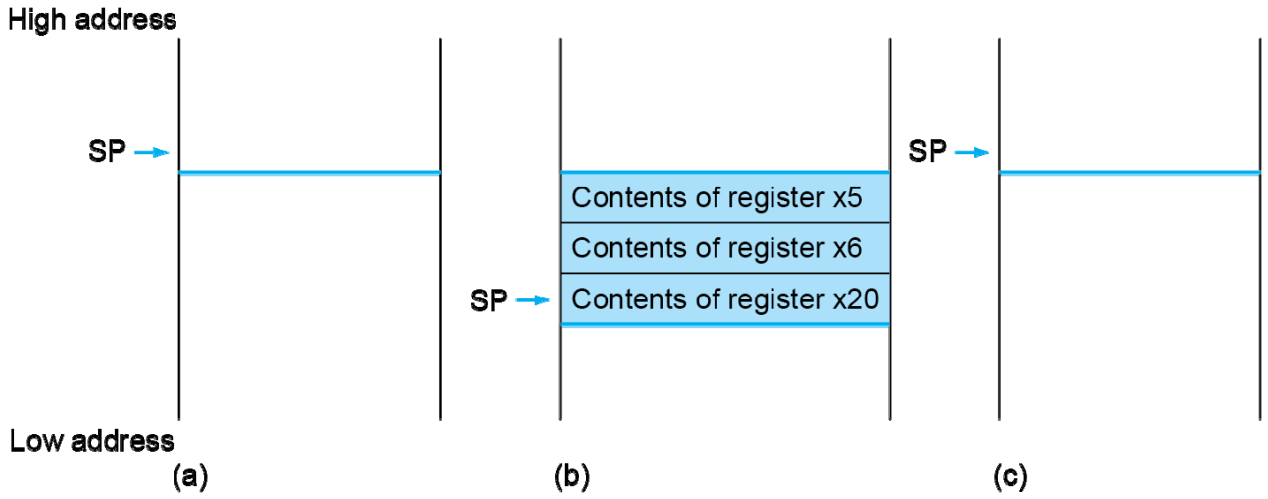
f = x5 – x6

copy f to return register

Resore x5, x6, x20 from stack

Return to caller

Local Data on the Stack



Register Usage

- x5 – x7, x28 – x31: temporary registers
  - Not preserved by the callee
- x8 – x9, x18 – x27: saved registers
  - If used, the callee saves and restores them

Non-Leaf Procedures

- Procedures that call other procedures
- For nested call, caller needs to save on the stack:
  - Its return address
  - Any arguments and temporaries needed after the call
- Restore from the stack after the call

Non-Leaf Procedure Example

- C code:

```
long long int fact (long long int n)
{
  if (n < 1) return f;
  else return n * fact(n - 1);
}
```

  - Argument n in x10
  - Result in x10

Leaf Procedure Example

- RISC-V code:

```
fact:
    addi sp,sp,-16
    sd   x1,8(sp)
    sd   x10,0(sp)
    addi x5,x10,-1
    bge  x5,x0,L1
    addi x10,x0,1
    addi sp,sp,16
    jalr x0,0(x1)
L1: addi x10,x10,-1
    jal  x1,fact
    addi x6,x10,0
    ld   x10,0(sp)
    ld   x1,8(sp)
    addi sp,sp,16
    mul  x10,x10,x6
    jalr x0,0(x1)
```

Save return address and n on stack

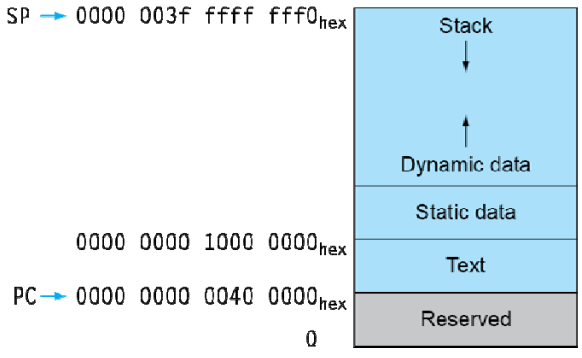
x5 = n - 1
if n >= 1, go to L1
Else, set return value to 1
Pop stack, don't bother restoring values
Return
n = n - 1
call fact(n-1)
move result of fact(n - 1) to x6
Restore caller's n
Restore caller's return address

Pop stack
return n * fact(n-1)

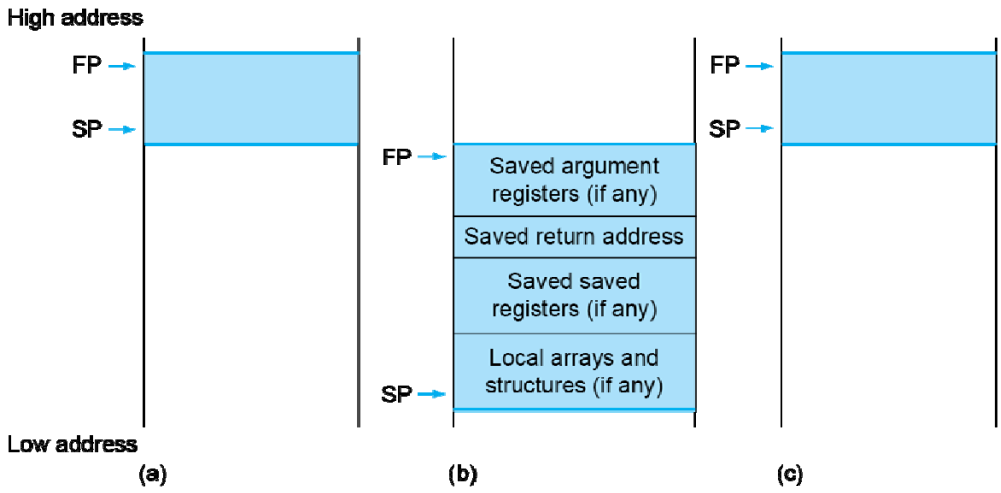
return

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - x3 (global pointer) initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage



Local Data on the Stack



- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

32-bit Constants

- Most constants are small
  - 12-bit immediate is sufficient
- For the occasional 32-bit constant

  `lui rd, constant`
  - Copies 20-bit constant to bits [31:12] of rd
  - Extends bit 31 to bits [63:32]
  - Clears bits [11:0] of rd to 0

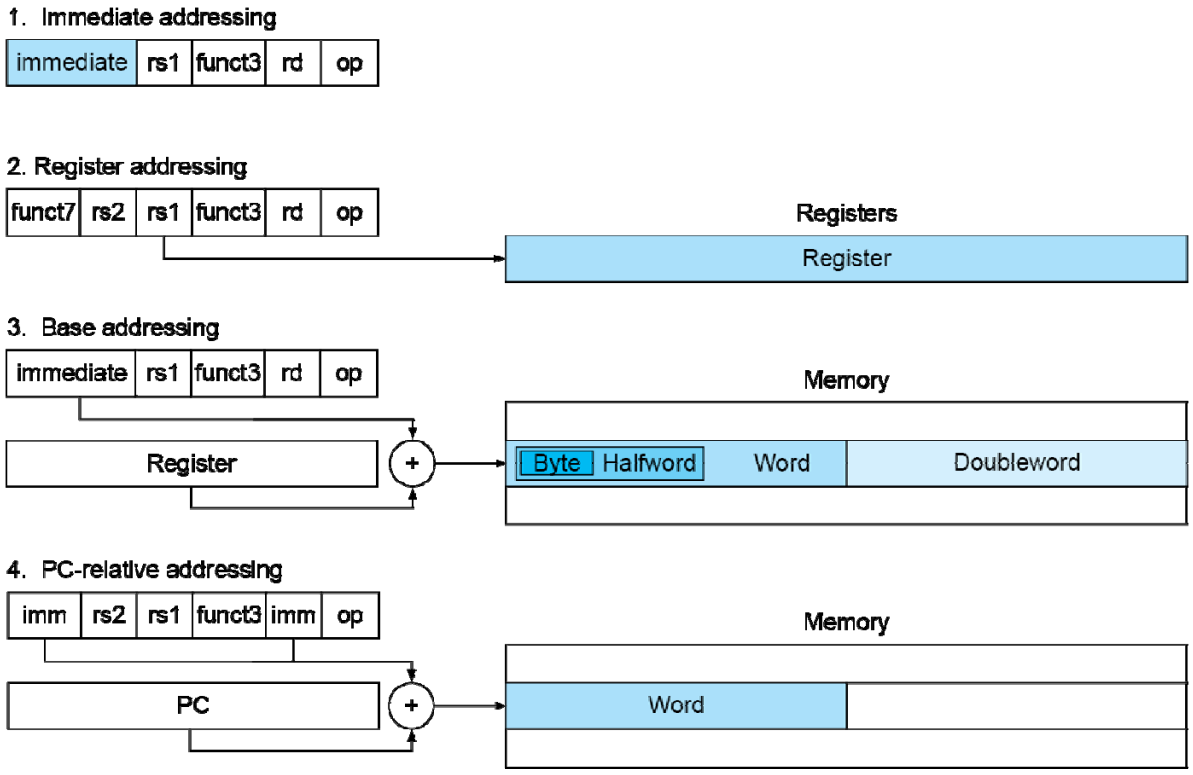
`lui x19, 976  // 0x003D0`

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0000 0000 0000

`addi x19,x19,128  // 0x500`

0000 0000 0000 0000	0000 0000 0000 0000	0000 0000 0011 1101 0000	0101 0000 0000

RISC-V Addressing Summary



Arrays vs. Pointers

- Array indexing involves
  - Multiplying index by element size
  - Adding to array base address
- Pointers correspond directly to memory addresses
  - Can avoid indexing complexity
- Multiply “strength reduced” to shift
- Array version requires shift to be inside loop
  - Part of index calculation for incremented i
  - c.f. incrementing pointer
- Compiler can achieve same effect as manual use of pointers
  - Induction variable elimination
  - Better to make program clearer and safer

Example: Clearing an Array

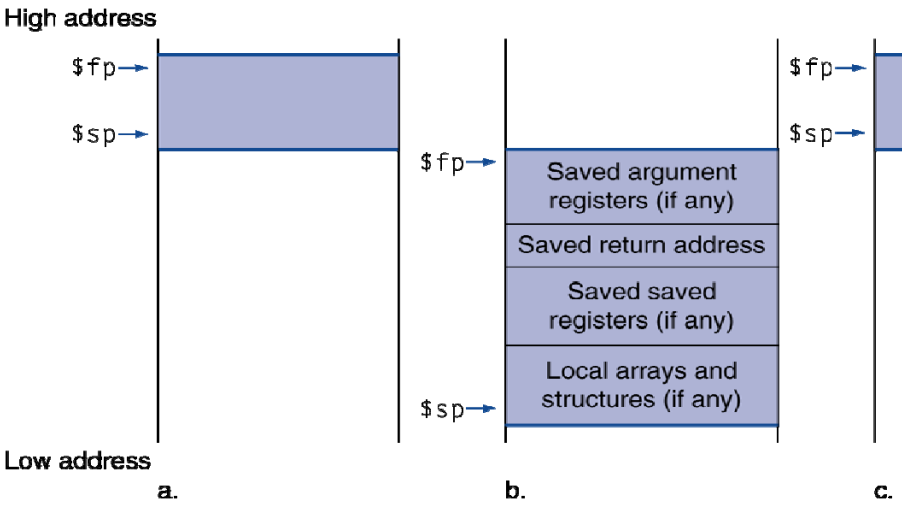
```
clear1(int array[], int size) {
  int i;
  for (i = 0; i < size; i += 1)
    array[i] = 0;
}
```

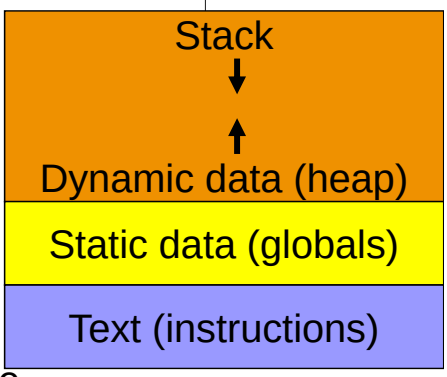
```
clear2(int *array, int size) {
  int *p;
  for (p = &array[0]; p < &array[size];
       p = p + 1)
    *p = 0;
}
```

```
   li   x5,0       // i = 0
loop1:
   slli x6,x5,3    // x6 = i * 8
   add  x7,x10,x6  // x7 = address
                   // of array[i]
   sd   x0,0(x7)   // array[i] = 0
   addi x5,x5,1    // i = i + 1
   blt  x5,x11,loop1  // if (i<size)
                      // go to loop1
```

```
   mv x5,x10       // p = address
                   // of array[0]
   slli x6,x11,3   // x6 = size * 8
   add x7,x10,x6   // x7 = address
                   // of array[size]
loop2:
   sd x0,0(x5)     // Memory[p] = 0
   addi x5,x5,8    // p = p + 8
   bltu x5,x7,loop2
                   // if (p<&array[size])
                   // go to loop2
```


Local Data on the Stack

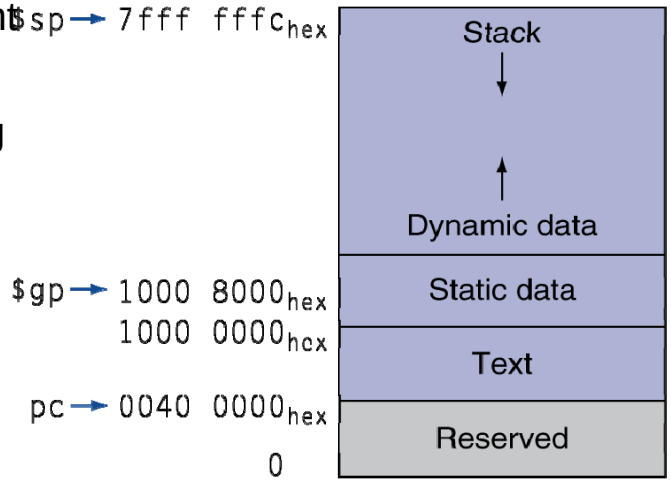




- Local data allocated by callee
  - e.g., C automatic variables
- Procedure frame (activation record)
  - Used by some compilers to manage stack storage

Memory Layout

- Text: program code
- Static data: global variables
  - e.g., static variables in C, constant arrays and strings
  - $gp initialized to address allowing ±offsets into this segment
- Dynamic data: heap
  - E.g., malloc in C, new in Java
- Stack: automatic storage

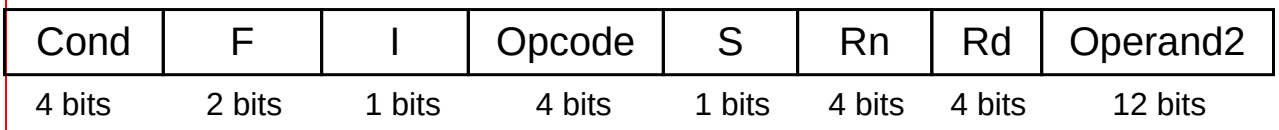


ARM Instruction Set

The ARM Instruction Set

- Most popular 32-bit instruction set in the world (www.arm.com)
- 4 Billion shipped in 2008
- Large share of embedded core market
  - Applications include mobile phones, consumer electronics, network/storage equipment, cameras, printers, …
- Typical of many modern RISC ISAs
- ARM has a 16 × 32-bit register file
  - Use for frequently accessed data
  - Registers numbered 0 to 15 (r0 to r15)
  - 32-bit data called a “word”

ARM Data Processing (DP) Instructions



- Instruction fields
  - Opcode : Basic operation of the instruction
  - Rd: The destination register operand
  - Rn: The first register source operand
  - Operand2: The second source operand
  - I:Immediate. If I is 0, the second source operand is a register, else the second source is a 12-bit immediate.
  - S: Set Condition Code
  - Cond: Condition
  - F: Instruction Format.

DP Instruction Example

Cond	F	I	Opcode	S	Rn	Rd	Operand2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

```
ADD r5,r1,r2 ; r5 = r1 + r2
```

14	0	0	4	0	1	5	2
4 bits	2 bits	1 bits	4 bits	1 bits	4 bits	4 bits	12 bits

$$1110000010000001010100000000010_2$$

ARM Data Transfer (DT) Instruction

Cond	F	Opcode	Rn	Rd	Offset12
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

LDR r5, [r3, #32] ; Temporary reg r5 gets A[8]

14	1	24	3	5	32
4 bits	2 bits	6 bits	4 bits	4 bits	12 bits

- **Design Principle 4**: Good design demands good compromises
  - Different formats complicate decoding, but allow 32-bit instructions uniformly
  - Keep formats as similar as possible

ARM instruction format summary

Name	Format	Example								Comments
Field size		4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	12 bits	All ARM instructions are 32 bits long
DP format	DP	Cond	F	I	Opcode	S	Rn	Rd	Operand2	Arithmetic instruction format
DT format	DT	Cond	F	Opcode			Rn	Rd	Offset12	Data transfer format
Field size		4 bits	2 bits	2 bits	24 bits					
BR format	BR	Cond	F	Opcode	signed_immed_24					B and BL instructions

Name	Register number	Usage	Preserved on call?
a1-a2	0–1	Argument / return result / scratch register	no
a3-a4	2–3	Argument / scratch register	no
v1-v8	4–11	Variables for local routine	yes
ip	12	Intra-procedure-call scratch register	no
sp	13	Stack pointer	yes
lr	14	Link Register (Return address)	yes
pc	15	Program Counter	n.a.

32-bit Constants

- Most constants are small
  - 16-bit immediate is sufficient
- For the occasional 32-bit constant
- Load 32 bit constant to r4

0000 0000 1101 1001 0000 0000 0000 0000

Cond	F	I	Opcode	S	Rn	Rd	rotate-imm	mm_8
14	0	1	13	0	0	4	8	217
4 bits	2 bits	1 bit	4 bits	1 bit	4 bits	4 bits	4 bits	8 bits

The 8 non-zerobits ($1101\ 1101_2$, 217_{ten}) of the constant is rotated by 16 bits and MOV instruction (opcode -13) loads the 32 bit value

Branch Instruction format

Condition	12	address
4 bits	4 bits	24 bits

- Encoding of options for Condition field

Value	Meaning	Value	Meaning
0	EQ (EQual)	8	HI (unsigned HIgher)
1	NE (Not Equal)	9	LS (unsigned Lower or Same)
2	HS (unsigned Higher or Same)	10	GE (signed Greater than or Equal)
3	LO (unsigned LOwer)	11	LT (signed Less Than)
4	MI (MInus, <0)	12	GT (signed Greater Than)
5	PL - (PLus, >=0)	13	LE (signed Less Than or Equal)
6	VS (oVerflow Set, overflow)	14	AL (Always)
7	VC (oVerflow Clear, no overflow)	15	NV (reserved)

ARM & MIPS Similarities

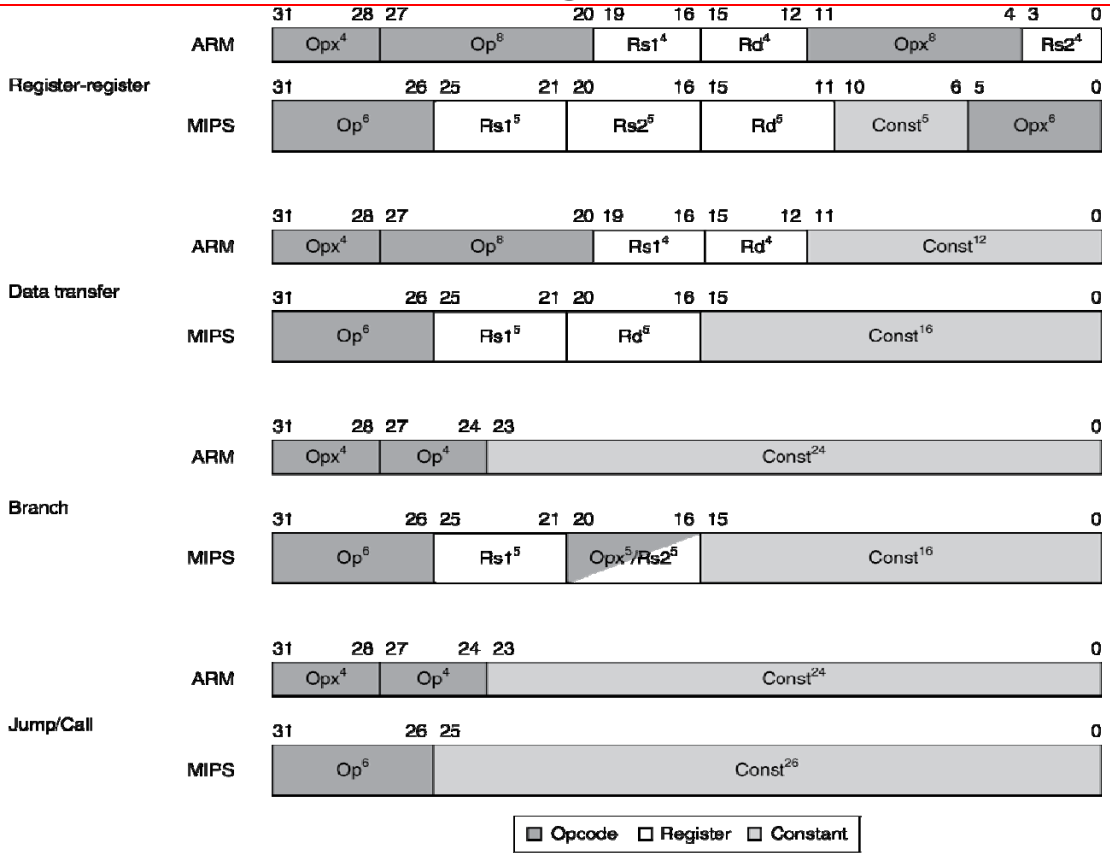
- ARM: the most popular embedded core
- Similar basic set of instructions to MIPS

	ARM	MIPS
Date announced	1985	1985
Instruction size	32 bits	32 bits
Address space	32-bit flat	32-bit flat
Data alignment	Aligned	Aligned
Data addressing modes	9	3
Registers	15 × 32-bit	31 × 32-bit
Input/output	Memory mapped	Memory mapped

Compare and Branch in ARM

- Uses condition codes for result of an arithmetic/logical instruction
  - Negative, zero, carry, overflow
  - Compare instructions to set condition codes without keeping the result
- Each instruction can be conditional
  - Top 4 bits of instruction word: condition value
  - Can avoid branches over single instructions

Instruction Encoding



IA-32

X86 Instruction set

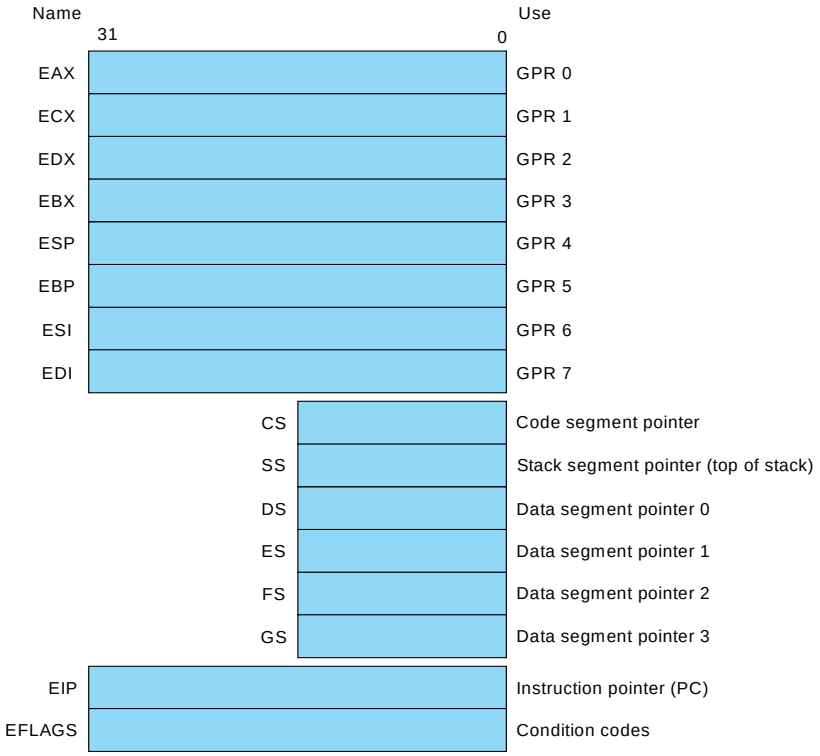
IA-32 Overview

- Complexity:
  - Instructions from 1 to 17 bytes long
  - one operand must act as both a source and destination
  - one operand can come from memory
  - complex addressing modes
    e.g., “base or scaled index with 8 or 32 bit displacement”
- Saving grace:
  - the most frequently used instructions are not too difficult to build
  - compilers avoid the portions of the architecture that are slow

“what the 80x86 lacks in style is made up in quantity, making it beautiful from the right perspective”

IA-32 Registers and Data Addressing

- Registers in the 32-bit subset that originated with 80386



IA-32 Register Restrictions

- Registers are not “general purpose” – note the restrictions below

Mode	Description	Register restrictions	MIPS equivalent
Register Indirect	Address is in a register.	not ESP or EBP	`lw $s0,0($s1)`
Based mode with 8- or 32-bit displacement	Address is contents of base register plus displacement.	not ESP or EBP	`lw $s0,100($s1)#≤16-bit #displacement`

Base plus scaled index	The address is Base + (2^{Scale} x Index) where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,0($t0)`

Base plus scaled index with 8- or 32-bit displacement	The address is Base + (2^{Scale} x Index) + displacement where Scale has the value 0, 1, 2, or 3.	Base: any GPR Index: not ESP	`mul $t0,$s2,4`
			`add $t0,$t0,$s1`
			`lw $s0,100($t0)#≤16-bit #displacement`

FIGURE 2.42 IA-32 32-bit addressing modes with register restrictions and the equivalent MIPS code. The Base plus Scaled Index addressing mode, not found in MIPS or the PowerPC, is included to avoid the multiplies by four (scale factor of 2) to turn an index in a register into a byte address (see Figures 2.34 and 2.36). A scale factor of 1 is used for 16-bit data, and a scale factor of 3 for 64-bit data. Scale factor of 0 means the address is not scaled. If the displacement is longer than 16 bits in the second or fourth modes, then the MIPS equivalent mode would need two more instructions: a `lui` to load the upper 16 bits of the displacement and an `add` to sum the upper address with the base register `$s1`. (Intel gives two different names to what is called Based addressing mode—Based and Indexed—but they are essentially identical and we combine them here.)

IA-32 Typical Instructions

- Four major types of integer instructions:
  - Data movement including move, push, pop
  - Arithmetic and logical (destination register or memory)
  - Control flow (use of condition codes / flags)
  - String instructions, including string move and string compare

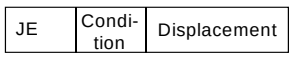
Instruction	Function
JE name	if equal(condition code) {EIP=name}; EIP-128 ≤ name < EIP+128
JMP name	EIP=name
CALL name	SP=SP-4; M[SP]=EIP+5; EIP=name;
MOVW EBX,[EDI+45]	EBX=M[EDI+45]
PUSH ESI	SP=SP-4; M[SP]=ESI
POP EDI	EDI=M[SP]; SP=SP+4
ADD EAX,#6765	EAX= EAX+6765
TEST EDX,#42	Set condition code (flags) with EDX and 42
MOVSL	M[EDI]=M[ESI]; EDI=EDI+4; ESI=ESI+4

FIGURE 2.43 Some typical IA-32 instructions and their functions. A list of frequent operations appears in Figure 2.44. The CALL saves the EIP of the next instruction on the stack. (EIP is the Intel PC.)

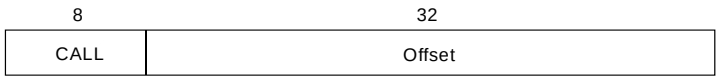
IA-32 instruction Formats

- Typical formats: (notice the different lengths)

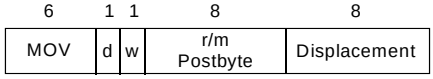
a. JE EIP + displacement



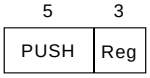
b. CALL



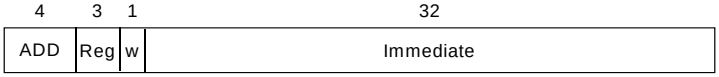
c. MOV EBX, [EDI + 45]



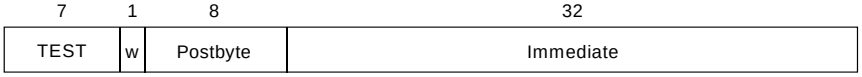
d. PUSH ESI



e. ADD EAX, #6765



f. TEST EDX, #42



Summary

- Instruction complexity is only one variable
  - lower instruction count vs. higher CPI / lower clock rate
- Design Principles:
  - simplicity favors regularity
  - smaller is faster
  - good design demands compromise
  - make the common case fast
- Instruction set architecture
  - a very important abstraction indeed!