Instruction Set Principles and Examples

(Appendix A)

Outline

- Classifying Instruction Set Architecture
- Memory addressing
- Type and size of Operands
- Operations in the Instruction Set
- Instructions for Control Flow
- Encoding an Instruction Set
- Crosscutting Issues: The Role of Compilers
- Putting It All Together: The RISC-V Architecture

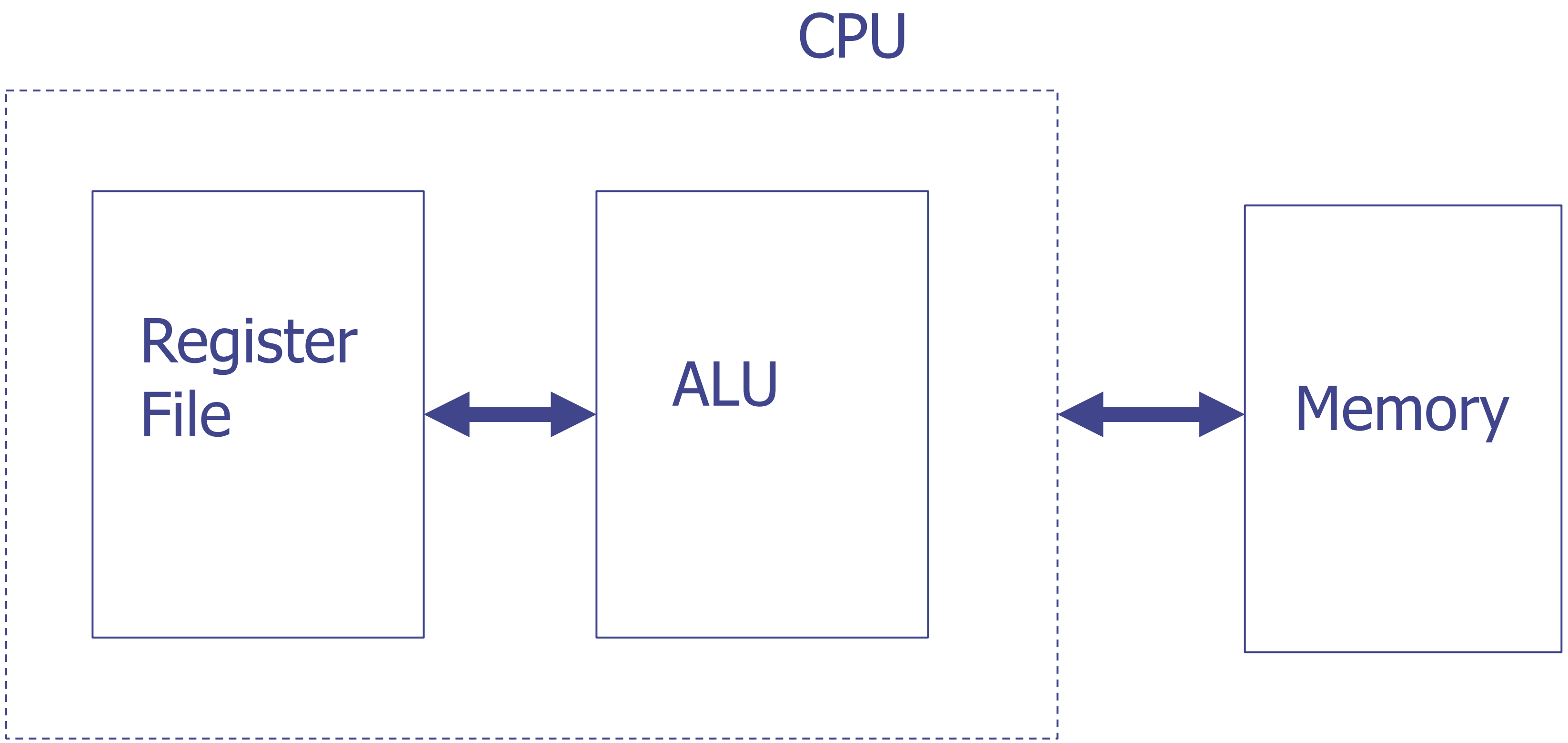
Classifying Instruction Set Architectures

There are two major components in a CPU:
register file and Arithmetic and Logic Unit (ALU).

A register file is used to hold the operands (source data for computation) and computation results.

A register file can be a stack, an accumulator or a set of general purpose registers.

An ALU is used to carry out the computation. It takes source data from the register file or external memory. Its computation results are also stored back to register file or external memory.


CPU
Register File
ALU
Memory

There are four instruction set architecture (ISA) classes:

(a) Stack: for CPU with a stack.

(b) Accumulator: for CPU with an accumulator.

(c) Register-Memory: for CPU with general purpose registers.

(d) Register-Register/Load-Store: for CPU with general purposed registers.

Stack

- The first operand is pointed by the Top of Stack (TOS) register.
- The second operand is right below the first operand.
- The result takes place of the second operand, and the first operand is removed from the stack.
- The TOS is updated to point to the second operand.

All operands are implicit for ALU instructions. Data can be transferred between stack and memories via instructions **push** and **pop**.

- Accumulator

The accumulator is both an implicit input and a result.

- Register-Memory

One input operand is a register, one is in memory, and the result goes to a register.

- Register-register/Load-store

All operands are registers. The results goes to a register. Data can be transferred between registers and memories via instructions **load** and **store**.

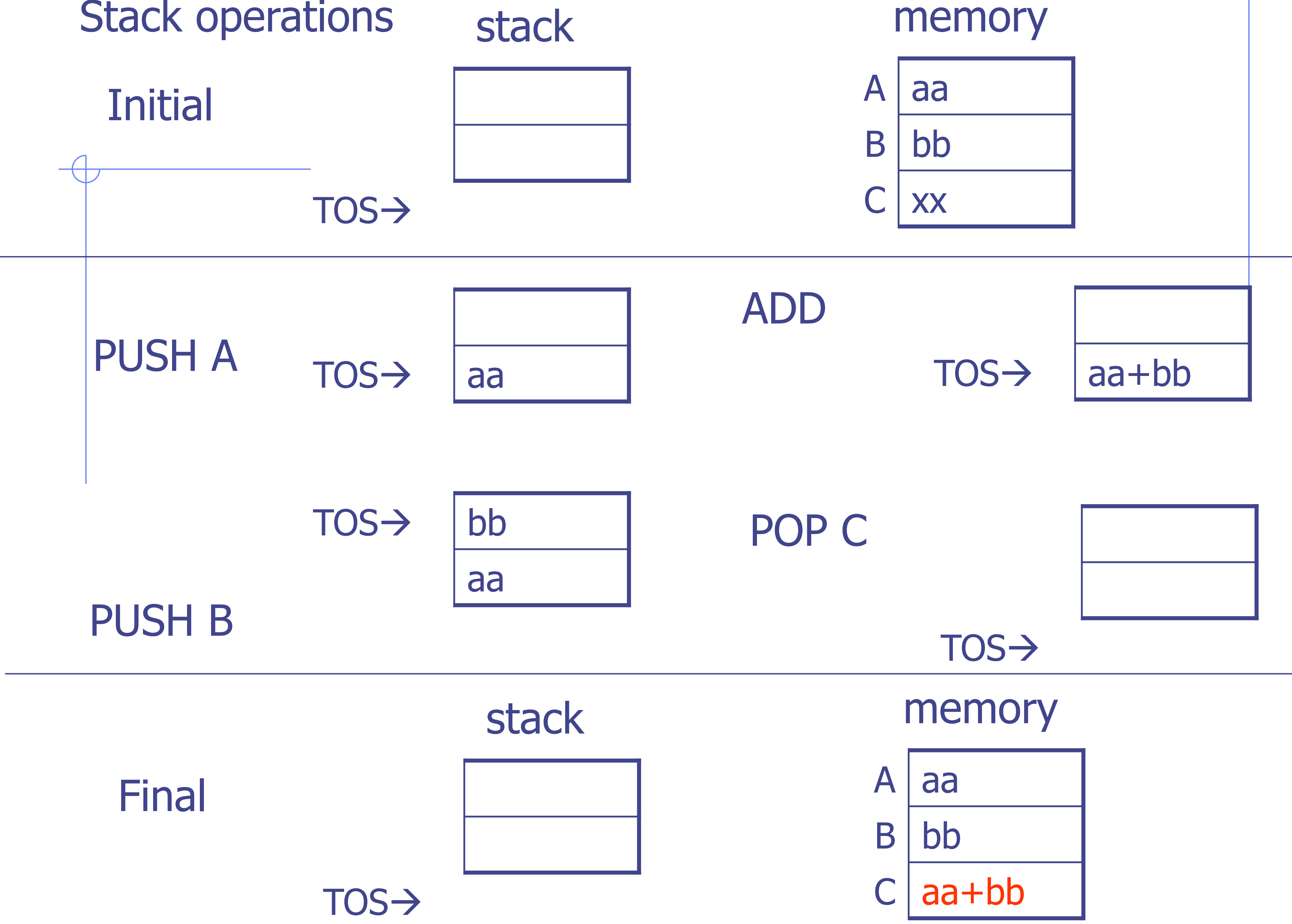
Example:

The code sequence for C=A+B for four classes of instruction set.

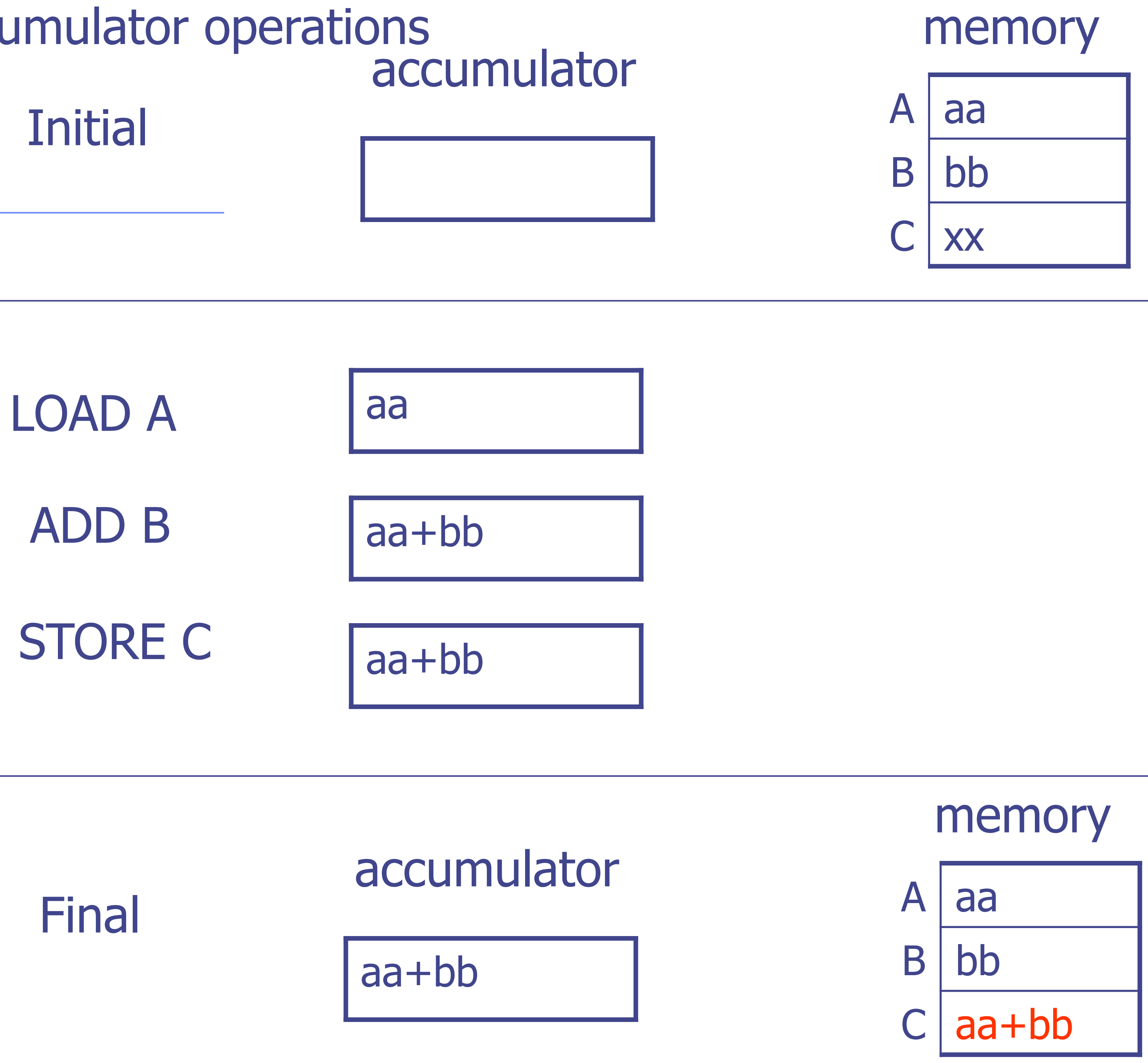
Stack	Accumulator	Register-Memory	Register-Register
Push A Push B Add Pop C	Load A Add B Store C	Load R1, A Add R3,R1,B Store R3,C	Load R1,A Load R2,B Add R3,R1,R2 Store R3,C

Here we assume A, B and C all belong in memory.

Stack operations



Accumulator operations



Resister-Memory

Initial

Register

R1	
R2	
R3	

memory

A	aa
B	bb
C	xx

LD R1,A

R1	aa
R2	
R3	

ADD R3,R1,B

R1	aa
R2	
R3	aa+bb

Store R3, C

R1	aa
R2	
R3	aa+bb

Final

Register

R1	aa
R2	
R3	aa+bb

memory

A	aa
B	bb
C	aa+bb

Resister-Register

Initial

Register

R1	
R2	
R3	

memory

A	aa
B	bb
C	xx

LD R1,A

R1	aa
R2	
R3	

LD R2, B

R1	aa
R2	bb
R3	

ADD R3, R2,R1

R1	aa
R2	bb
R3	aa+bb

Store R3, C

R1	aa
R2	bb
R3	aa+bb

Final

Register

R1	aa
R2	bb
R3	aa+bb

memory

A	aa
B	bb
C	aa+bb

◆General-purpose register (GPR) computers

GPR computers
- Register-memory
- Register-register (load-store)
- Memory-memory (not used today)

Although most early computers used stack or accumulator-style architectures, virtually new architecture designed after 1980 uses a load-store register architecture.

There are two main reasons for using the GPR computers:

1. Registers are faster than memory.
2. Registers are more efficient for a compiler to use than other forms of internal storage.

Example: (A*B)-(B*C)-(A*D)

For GPR computers, the multiplication can be performed in any order, which may be more efficient because of the location of operands or because of the pipeline concerns.

For stack computers, the multiplication can only be performed in single order, and it may have to load an operand multiple times.

Two major instruction set characteristics divides GPR architectures.

1. Maximum number of operands allowed for ALU instructions (2 or 3).
2. Number of memory operands for ALU instructions (0,1,2 or 3).

Examples of the GPR computers are shown below.

Number of Memory Addresses	Maximum Number of Operands	Type of Architecture	Examples
0	3	Load-Store	ARM, MIPS, PowerPC, SPARC, RISC-V
1	2	Register-memory	Intel 80x86, Motorola 680000, TI TMS320C54
2	2	Memory-Memory	VAX
3	3	Memory-Memory	VAX

Memory Addressing

- Interpreting memory addresses

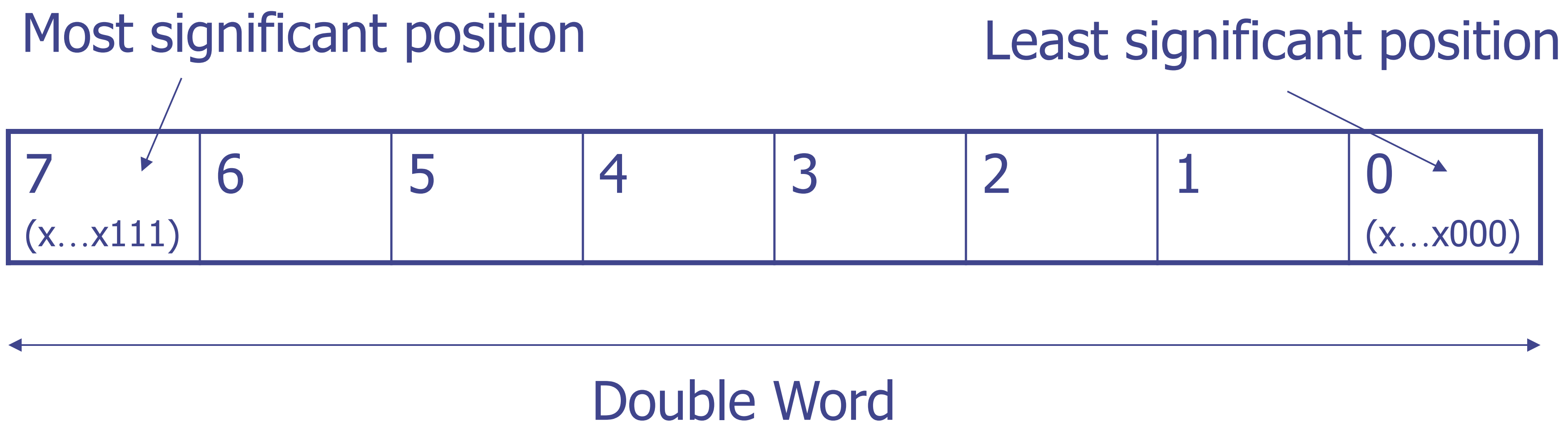
Here we study what object is accessed as the function of address and length.

All instruction sets discussed in this course are byte-addressed and provide access for bytes (8 bits), half words (16 bits), words (32 bits) and double words (64 bits).

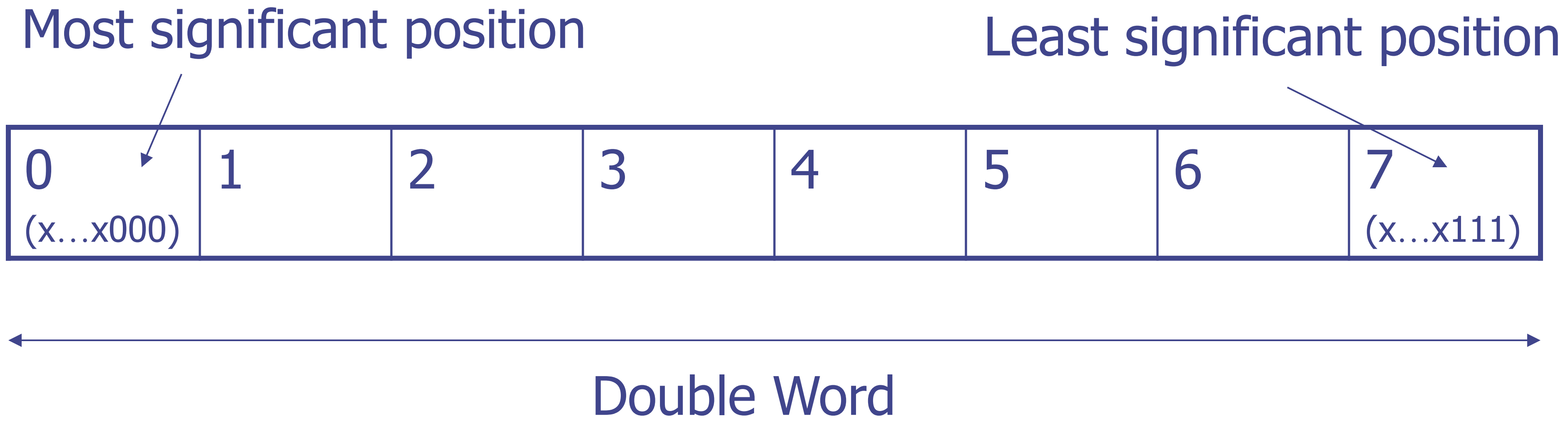
•Little Endian and Big Endian

There are two different conventions for ordering bytes within a larger object: little endian and big endian.

Little endian byte order puts the byte whose address is "x…x000" at the least-significant position in the double word.



Big endian byte order puts the byte whose address is "x…x000" at the most-significant position in the double word.



- Aligned memory access

An access to an object of size s bytes at byte address A is aligned if

$$A \bmod s = 0.$$

Addressing Modes

The following shows the most common names for the addressing modes.

Addressing Mode	Example	Meaning
Register	ADD R4,R4,R3	Reg[R4]$\leftarrow$Reg[R4]+Reg[R3]
Immediate	ADD R4,R4,3	Reg[R4]$\leftarrow$Reg[R4]+3
Displacement	ADD R4,R4,100(R1)	Reg[R4]$\leftarrow$Reg[R4]+MEM[100+Reg[R1]]
Register Indirect	ADD R4,R4,(R1)	Reg[R4]$\leftarrow$Reg[R4]+MEM[REG[R1]]
Indexed	ADD R4,R4,(R1+R2)	Reg[R4]$\leftarrow$Reg[R4]+ MEM[Reg[R1]+Reg[R2]]
Memory Indirect	ADD R4,R4,@(R3)	Reg[R4]$\leftarrow$Reg[R4]+ MEM[MEM[Reg[R3]]]
Scaled	ADD R4,R4,100(R1)[R2]	Reg[R4]$\leftarrow$Reg[R4]+ MEM[100+Reg[R1]+Reg[R2]*d]

Addressing modes have the ability to significantly **reduce instruction counts**; they also add to the complexity of building a computer and may **increase the average CPI**. Therefore, the usage of various addressing mode is quite important in helping the architect choose what to include.

We consider the results of measuring addressing mode usage patterns in three programs on the VAX architecture. The old VAX architecture is used here because it has the richest set of addressing modes.

Frequency of addressing mode for VAX architecture

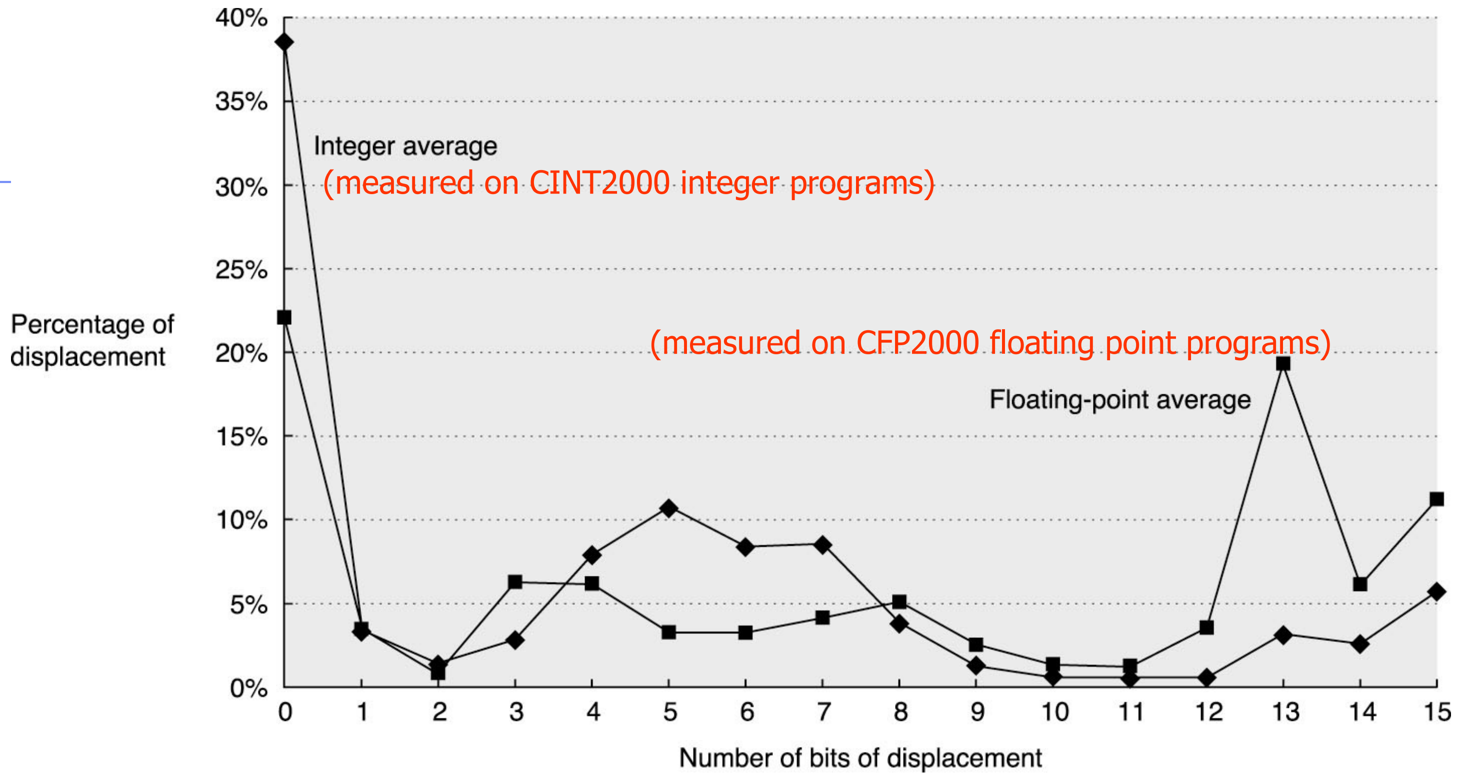
	Memory Indirect	Scaled	Register Indirect	Immediate	Displacement
tex	1%	0	24%	43%	32%
spice	6%	16%	3%	17%	55%
gcc	1%	6%	11%	39%	40%

As the table shows, immediate and displacement addressing dominate addressing mode usage.

- Displacement addressing mode

The range of displacement is important for the displacement addressing mode since it affects the instruction length.

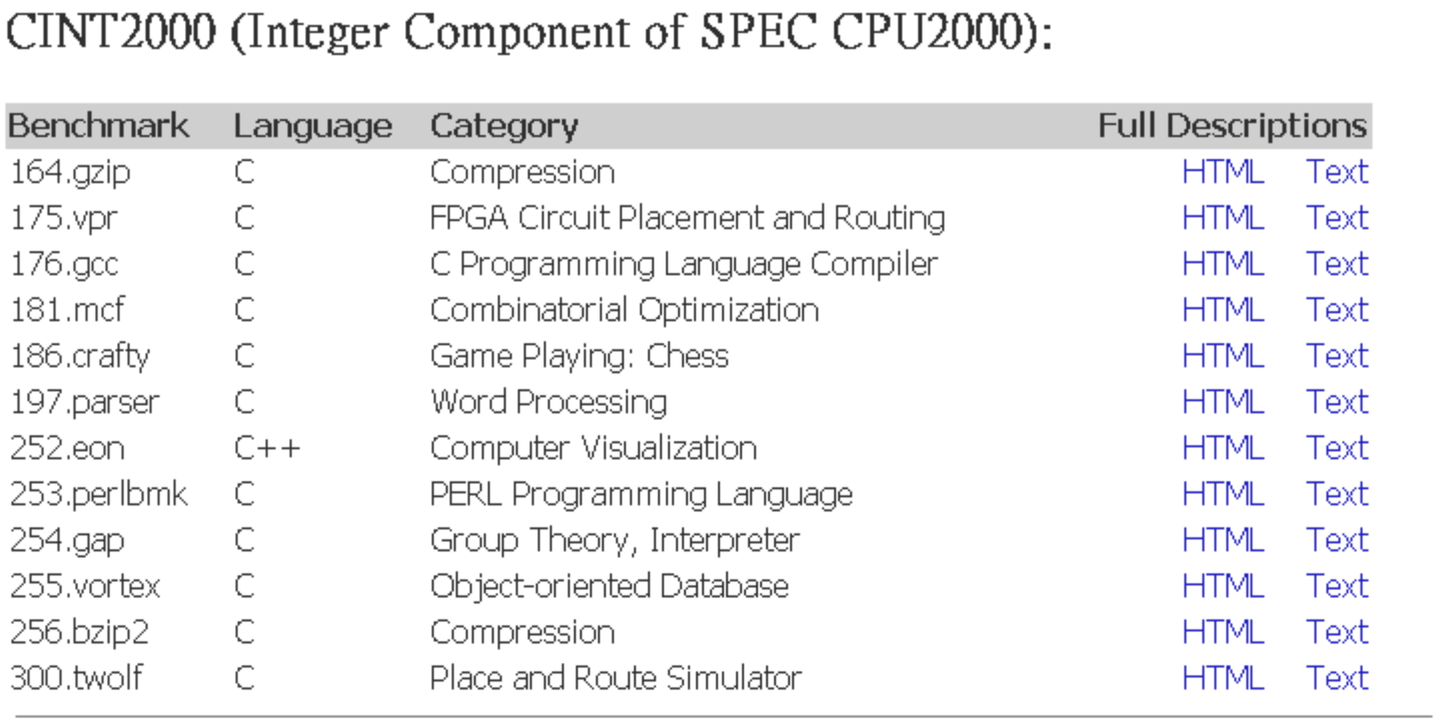
The following shows the measurements taken on the data access on a load-store architecture using the benchmark programs.



Alpha architecture
SPEC CPU2000

Note that the graph does not include the sign bit.
From the figure, we see that the displacement values
are widely distributed.

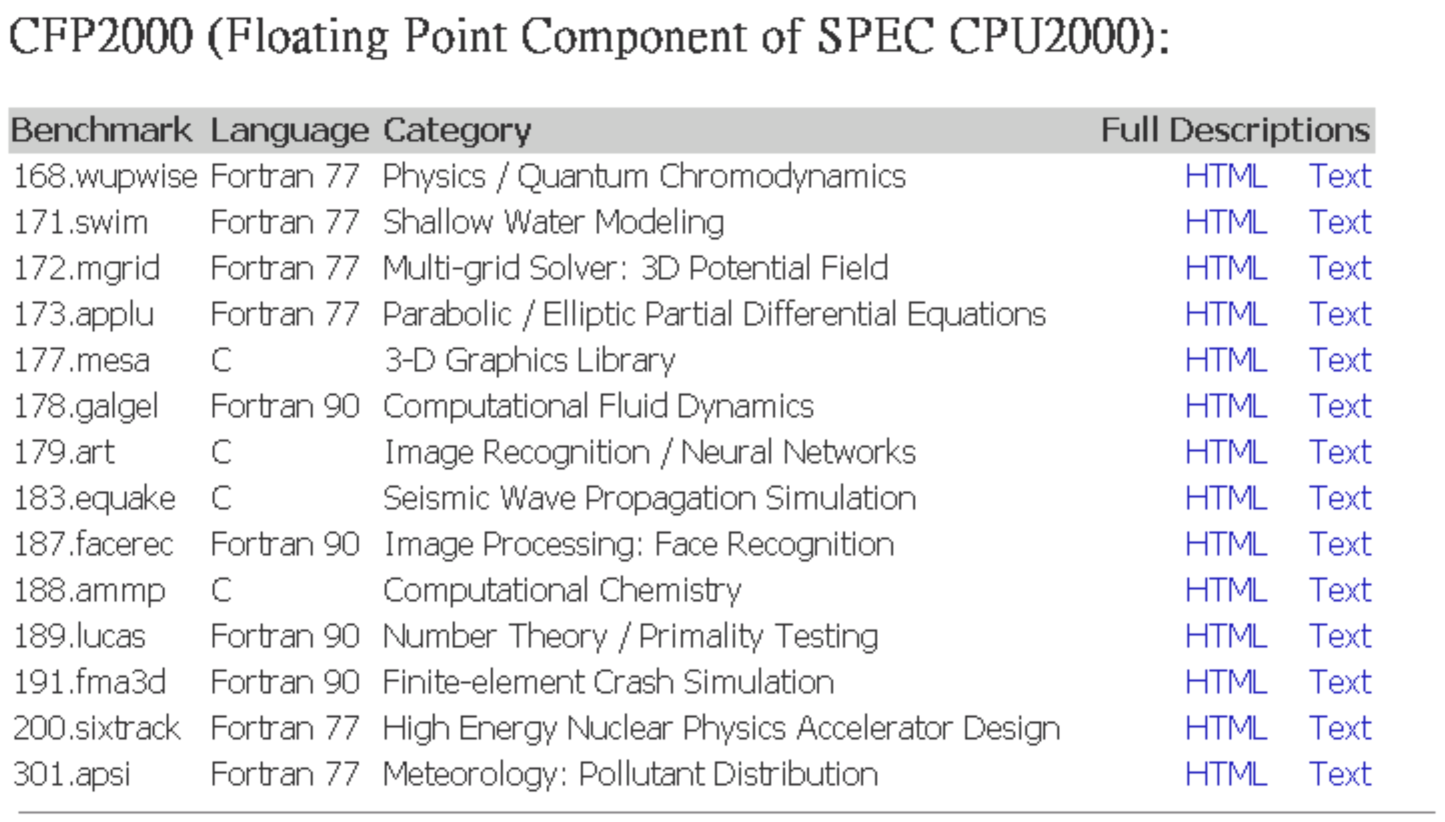
CINT2000 Benchmark



CINT2000 (Integer Component of SPEC CPU2000):

Benchmark	Language	Category	Full Descriptions	
164.gzip	C	Compression	HTML	Text
175.vpr	C	FPGA Circuit Placement and Routing	HTML	Text
176.gcc	C	C Programming Language Compiler	HTML	Text
181.mcf	C	Combinatorial Optimization	HTML	Text
186.crafty	C	Game Playing: Chess	HTML	Text
197.parser	C	Word Processing	HTML	Text
252.eon	C++	Computer Visualization	HTML	Text
253.perlbmk	C	PERL Programming Language	HTML	Text
254.gap	C	Group Theory, Interpreter	HTML	Text
255.vortex	C	Object-oriented Database	HTML	Text
256.bzip2	C	Compression	HTML	Text
300.twolf	C	Place and Route Simulator	HTML	Text

CFP2000 Benchmark



CFP2000 (Floating Point Component of SPEC CPU2000):

Benchmark	Language	Category	Full Descriptions	
168.wupwise	Fortran 77	Physics / Quantum Chromodynamics	HTML	Text
171.swim	Fortran 77	Shallow Water Modeling	HTML	Text
172.mgrid	Fortran 77	Multi-grid Solver: 3D Potential Field	HTML	Text
173.applu	Fortran 77	Parabolic / Elliptic Partial Differential Equations	HTML	Text
177.mesa	C	3-D Graphics Library	HTML	Text
178.galgel	Fortran 90	Computational Fluid Dynamics	HTML	Text
179.art	C	Image Recognition / Neural Networks	HTML	Text
183.equake	C	Seismic Wave Propagation Simulation	HTML	Text
187.facerec	Fortran 90	Image Processing: Face Recognition	HTML	Text
188.ammp	C	Computational Chemistry	HTML	Text
189.lucas	Fortran 90	Number Theory / Primality Testing	HTML	Text
191.fma3d	Fortran 90	Finite-element Crash Simulation	HTML	Text
200.sixtrack	Fortran 77	High Energy Nuclear Physics Accelerator Design	HTML	Text
301.apsi	Fortran 77	Meteorology: Pollutant Distribution	HTML	Text

- Immediate addressing mode

For the use of immediates it is important to know whether they need to be supported for all operations or for only a subset.

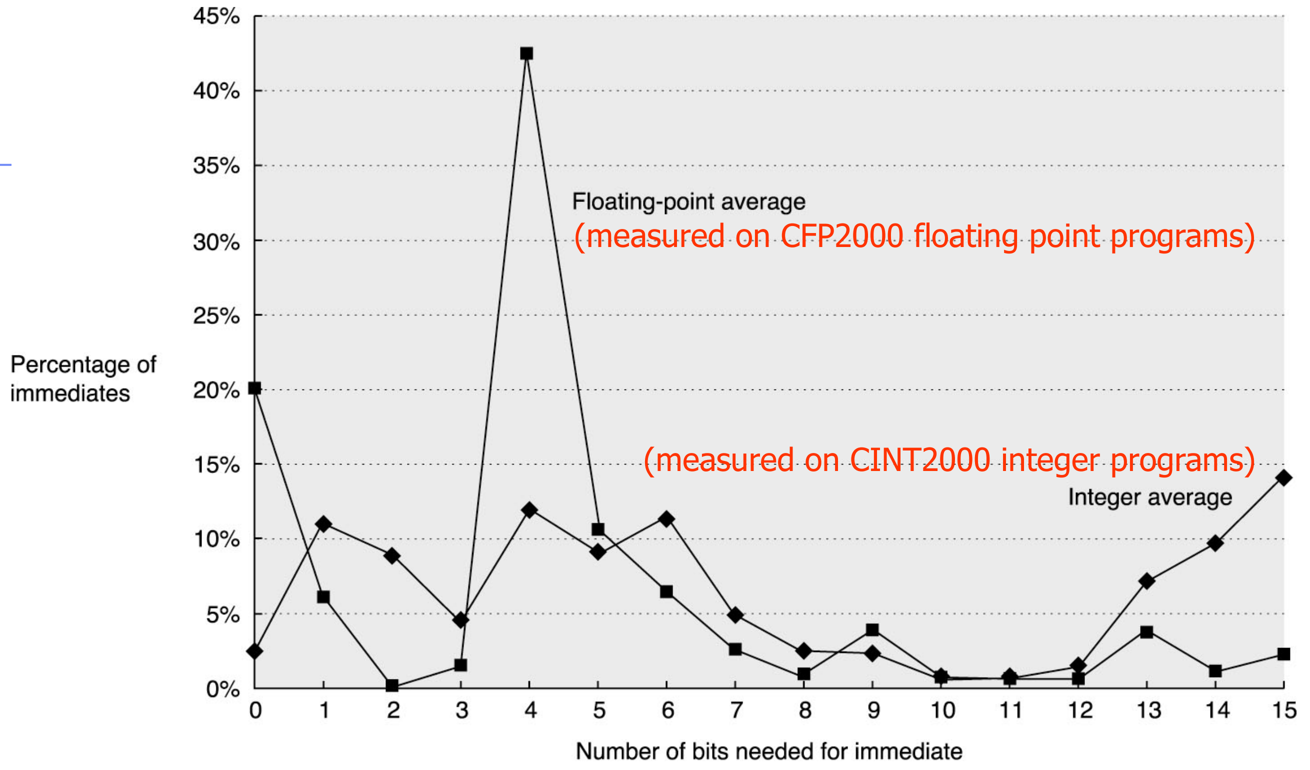
Another important instruction set measurement is the range of values for immediates.

Frequency of instructions using immediate values

Alpha architecture
SPEC CPU2000

	Loads	ALU Operations	All Instructions
Floating Point (CFP2000)	22%	19%	16%
Integer (CINT2000)	23%	25%	21%

Here we see that about ¼ of the loads and ALU operations have an immediate operands.



Alpha architecture
SPEC CPU2000

The figure shows that small immediate values are most heavily used. Large immediates are sometimes used, however, most likely in addressing calculations.

Type and size of operands

Common operand types include character (8 bits), half word (16 bits), word (32 bits), single-precision floating point (also 1 word), and double-precision floating point (2 words).

Integers are represented as 2's complement binary numbers. Characters are in ASCII or Unicode. Floating points are based on IEEE standard 754.

For business applications, some architectures support a decimal format, usually called packed decimal or binary coded decimal, where 4 bits are used to encode values 0-9, and 2 decimal digits are packed into each byte.

Operations in the instruction set

The operators supported by most instruction set architectures can be categorized as follows.

Operator type	Example
Arithmetic and logical	Integer arithmetic and logical operations: add, sub, and, or, mul and div
Data transfer	Load and store
Control	Branch, call and return
System	Operating system call, virtual memory management instructions
Floating point	Floating point operations: add, mul, div and compare
Decimal	Decimal add, decimal mul
String	String compare, string move
Graphics	Pixel and vertex operations

Rank	80x86 instructions	% of total executed
1	Load	22%
2	Conditional Branch	20%
3	Compare	16%
4	Store	12%
5	Add	8%
6	And	6%
7	Sub	5%
8	Move	4%
9	Call	1%
10	Return	1%
Total		96%

Here we shows 10 simple instructions that account for 96% of instructions executed for a collection of integer programs running on Intel 80x86. Hence, the implementation of these instructions should be sure to make these fast.

Instructions for control flow

There are four different types of instructions for flow control:

1. Conditional branches
2. Jump
3. Procedure calls
4. Procedure returns

Frequency of branch instructions

	Call/Return	Jump	Conditional Branch
Floating Point (CFP2000)	8%	10%	82%
Integer (CINT2000)	19%	6%	75%

Alpha architecture
SPEC CPU2000

It is clear from the figure that conditional branch is the most frequently used instruction.

Addressing Modes for Control Flow Instructions

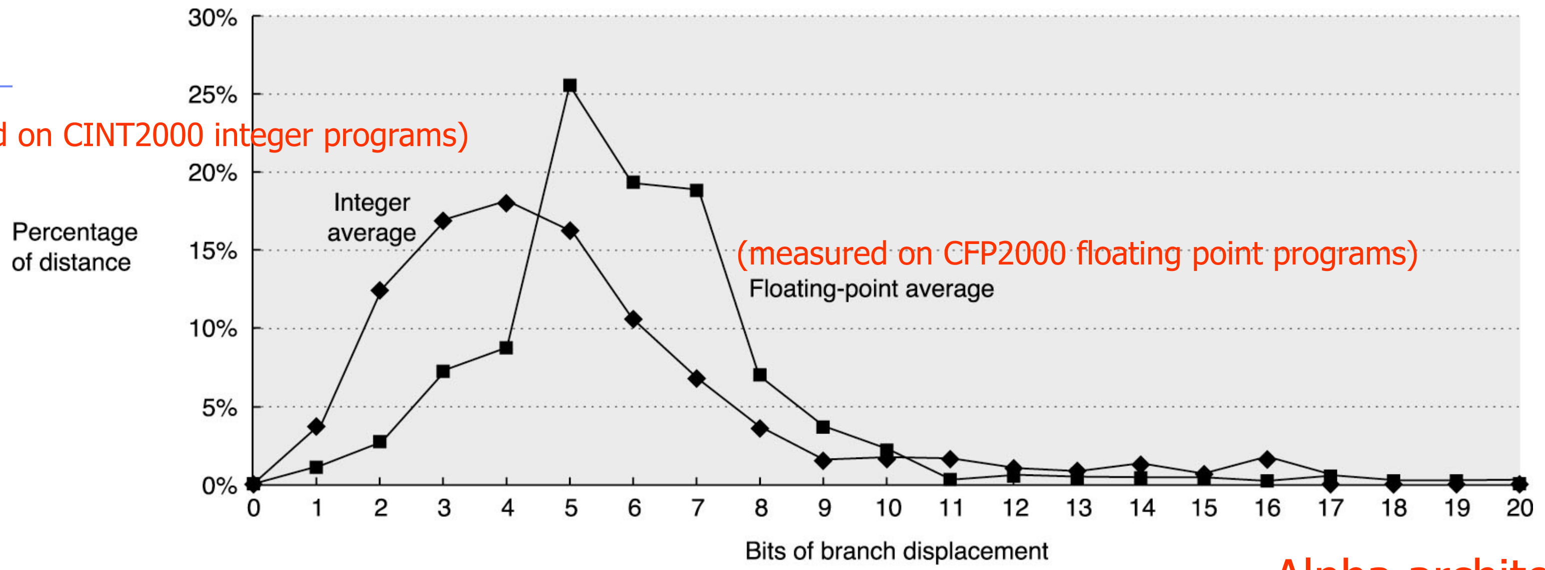
The addressing mode is dependent on whether the target address is available at compile time.

- Target address is available

In this case, we can specify the target address explicitly. The most common way to specify the destination is to supply the displacement that is added to the program counter (PC). Control flow instructions of this sort are called PC-relative.

PC-relative branches have the following advantages

1. Because the target is often near the current position, specifying the position relatively to the current PC requires fewer bits.
2. Using PC-relative addressing also permits the code to run independently of where it is loaded.



Alpha architecture
SPEC CPU2000

From the figure, we see that the most frequent branches in the integer programs are to targets that can be encoded in 4-8 bits.

- Target address is not available

To implement returns and indirect jumps when the target is not known at compile time, a method other then PC-relative addressing is required.

A common addressing mode for this case is the register-indirect mode, where the target address is contained in a register.

Conditional Branch Options

Since most changes in control flow are branches, deciding how to specify the branch condition is important.

There are three major approaches: Condition code, Condition register, Compare and branch

Name	CPU	Code Example	Advantages
Condition Code	80x86 ARM PowerPC SPARC	SUB R1,R2,R3 BNZ Target	Sometime condition is set for free
Condition Register	Alpha, MIPS	SUB R1,R2,R3 BNZ R1,Target	Simple
Compare and branch	RISC-V VAX	BNE R2,R3, Target	One instruction rather than two for a branch

The following shows the frequency of different comparisons for conditional branch.

Alpha architecture
SPEC CPU2000

	Not equal	Equal	Greater than or equal	Greater than	Less than or equal	Less than
Floating Point (CFP2000)	5%	16%	0%	0%	44%	34%
Integer (CINT2000)	2%	18%	11%	0%	33%	35%

From the figure, we see that *compare equal*, *compare not equal* and *compare less* account for approximately 90% of comparisons.

Encoding an instruction set

The architect must balance several competing forces when encoding the instruction set:

1. The desire to have as many registers and addressing modes as possible.
2. The impact of the size of the register and addressing mode fields on the average instruction size and hence on the average program size.
3. A desire to have instructions encoded into lengths that will be easy to handle in a pipelined implementation.

The following shows three popular choices for encoding the instruction set.

Operation and no. of operands	Address specifier 1	Address field 1	...	Address specifier n	Address field n

Variable (Intel 80x86, VAX)

Operation	Address field 1	Address field 2	Address field 3

Fixed (RISC V, ARM, Power PC)

Operation	Address specifier	Address field

Operation	Address specifier	Address field 1	Address field 2

Hybrid (RISC V Compressed, IBM 360/370, ARM Thumb2)

The variable format has the smallest program size.

The fixed format is well-suited for pipeline implementation. Therefore, it has the highest performance.

The hybrid format may have the advantages of both variable and fixed formats.

Crosscutting Issues: The role of Compilers

- Register allocation

Register allocation plays the central role both in speeding up the code and in making other optimization useful. Therefore, it is one of the the most important optimization for a compiler.

The register allocation algorithms today are based on a technique called graph coloring. The technique works best when there are at least **16** general-purpose registers available for global allocation for integer variables.

◆How the architect can help the compiler writer

- Provide regularity

The three primary components of an instruction set (the operations, the data types and the addressing modes) should be orthogonal (i.e., independent).

- Provide primitives, not solutions

Special features that “match” a language construct or a kernel function are often unusable.

Putting It All Together: The RISC-V Architecture

In this section we describe a simple 64-bit architecture called RISC-V, which emphasizes

1. A simple load-store instruction set
2. Design for pipelining efficiency
3. Efficiency as a compiler target

RISC-V Overview

The RISC-V instruction set is organized as three base instruction sets (namely RV32I, RV32E and RV64I), and a varieties of optional extensions.

Name	Functionality
RV32I	Base 32-bit integer instruction with 32 registers
RV32E	Base 32-bit integer instruction with 16 registers
RV64I	Base 64-bit instruction set; All registers are 64-bits
M	Integer multiply and divide
A	Automic instructions for concurrent operations
F	Single precision floating point instructions
D	Double precision floating point instructions
V	Vector operations
P	SIMD instructions

Both RV32I and RV32E are used only for small embedded processors.

We call the RV64I with extensions M, A, F, and D the RV64G.

That is, RV64G=RV64I+M+A+F+D.

RV64G can be used for a larger varieties of computers.

We consider only RV64G in this chapter.

Registers for RISC-V

RV64G has 32 64-bit general-purpose registers (GPRs), named x0,…,x31. GPRs are also sometimes known as integer registers.

RV64G also contains 32 floating-point registers (FPRs), names f0,…,f31, which can hold 32 single precision (32-bit) values or 32 double precision (64-bit) values.

The value of x0 is always zero.

Data types for RISC-V

The data types are 8-bit bytes, 16-bit half words, 32-bit words, and 64-bit double words for integer data and 32-bit single precision and 64-bit double precision for floating point.

Addressing modes for RISC-V data transfers

The only data addressing modes are **immediate** and **displacement**, both with 12-bit fields. Register indirect is accomplished by placing 0 in the 12-bit displacement field, and absolute addressing with a 12-bit field is accomplished by using x0 as the base register.

Memory access should be byte addressable with a 64-bit address. It uses Little Endian.

Memory accesses need not to be aligned. However, unaligned accesses run extremely slow.

The only operations that affect memory are **load** and **store**.

RISC-V Instruction Format

There are 4 RISC-V instruction formats: R-type, I-type S-type, and U-type. All the formats are 32-bit long with 7-bit primary opcode.

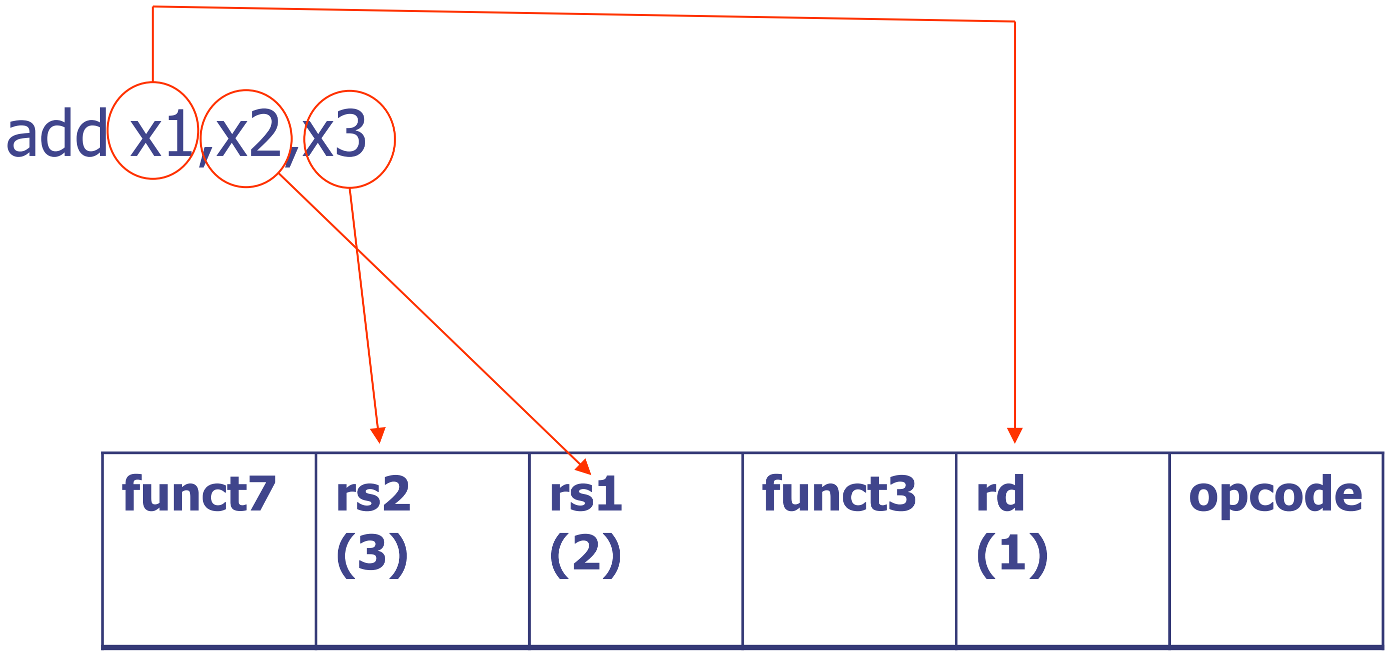
R-type, I-type and S-type are commonly used.

•R-type format

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

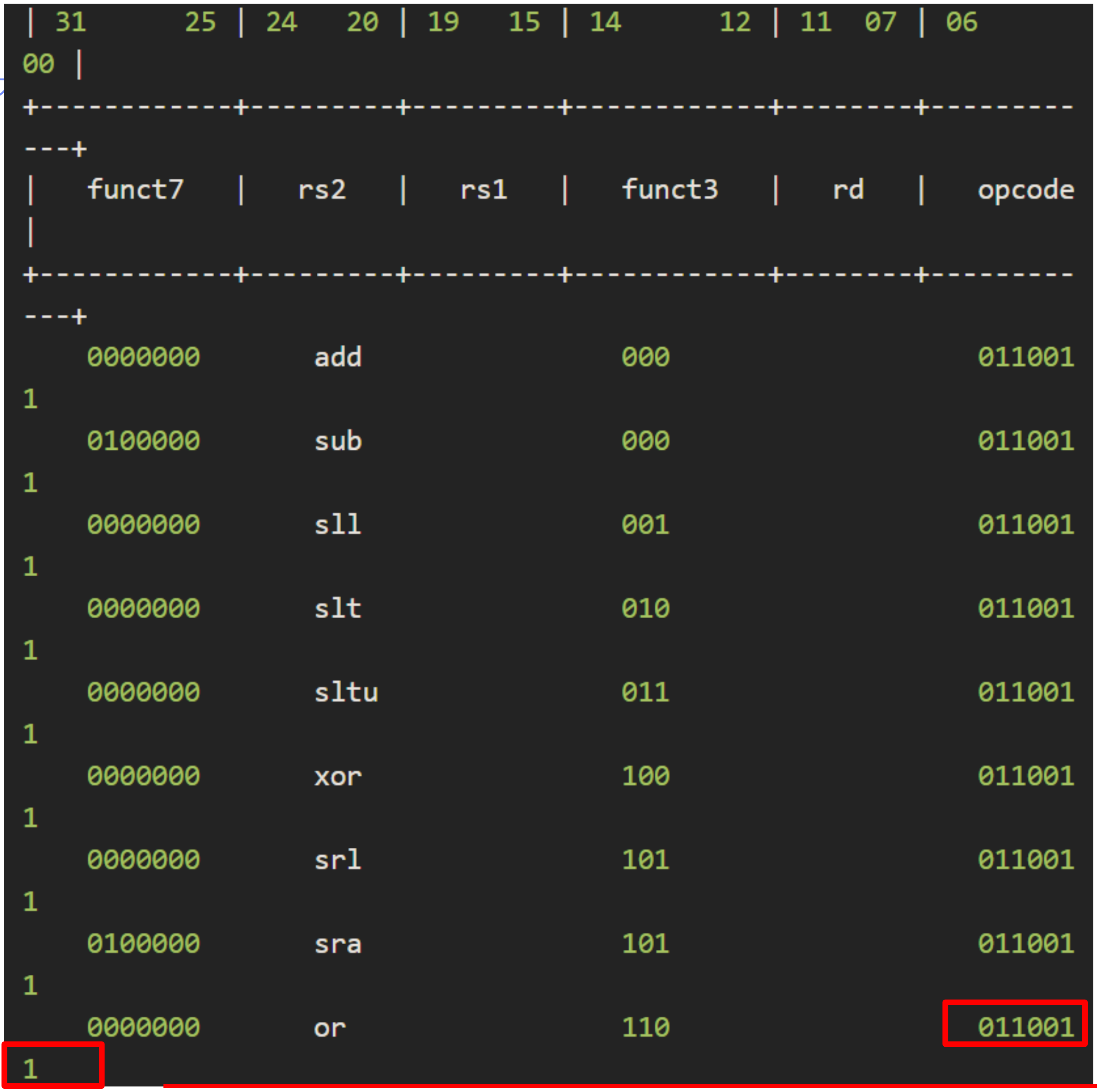
Register-Register ALU instructions.

Example:



Reg[x1] ← Reg[x2]+Reg[x3]

ALU instructions such as **add, sub, sll, srl, xor, or** share the same opcode=0110011, but have different **funct3** and **func7** values.



```
| 31       25 | 24    20 | 19    15 | 14       12 | 11   07 | 06
00 |
+------------+---------+---------+------------+--------+--------
---+
|   funct7   |   rs2   |   rs1   |   funct3   |   rd   |   opcode
|
+------------+---------+---------+------------+--------+--------
---+
    0000000       add                000                  011001
1
    0100000       sub                000                  011001
1
    0000000       sll                001                  011001
1
    0000000       slt                010                  011001
1
    0000000       sltu               011                  011001
1
    0000000       xor                100                  011001
1
    0000000       srl                101                  011001
1
    0100000       sra                101                  011001
1
    0000000       or                 110                  011001
1
```

$\boxed{011001}$ $\boxed{1}$ → Opcode=
0110011

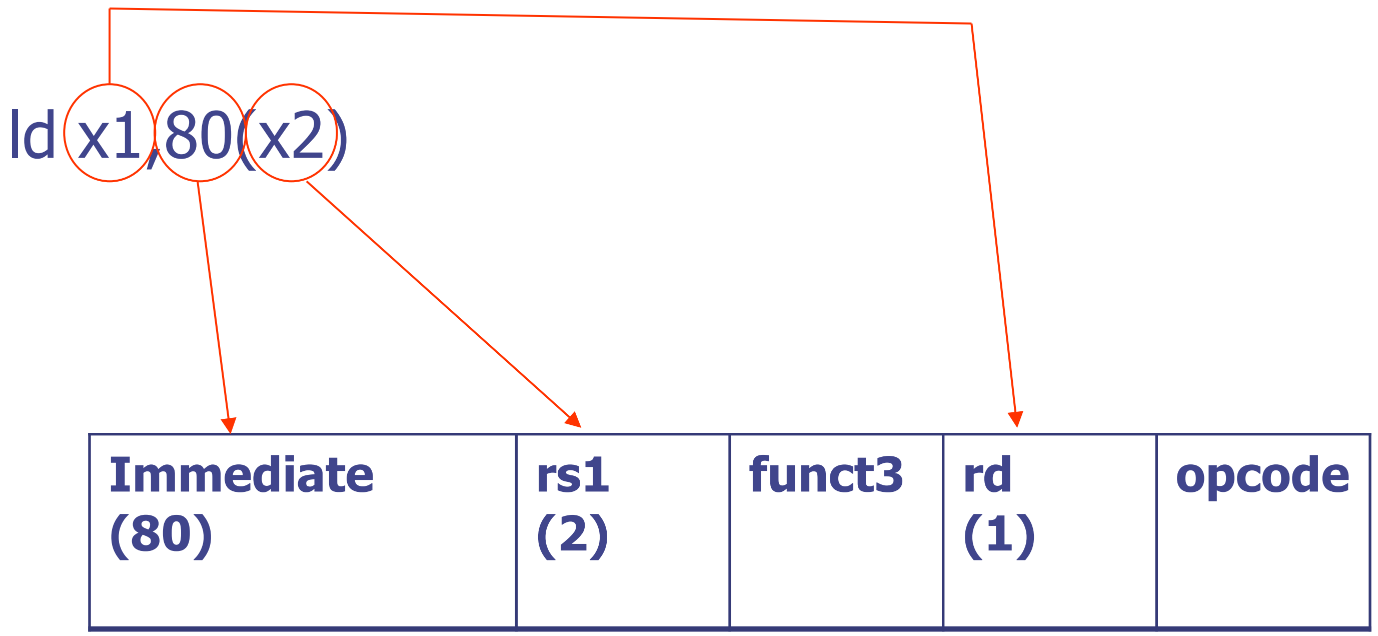
•I-type format

Immediate [11:0]	rs1	funct3	rd	opcode
12	5	3	5	7

Load instructions.

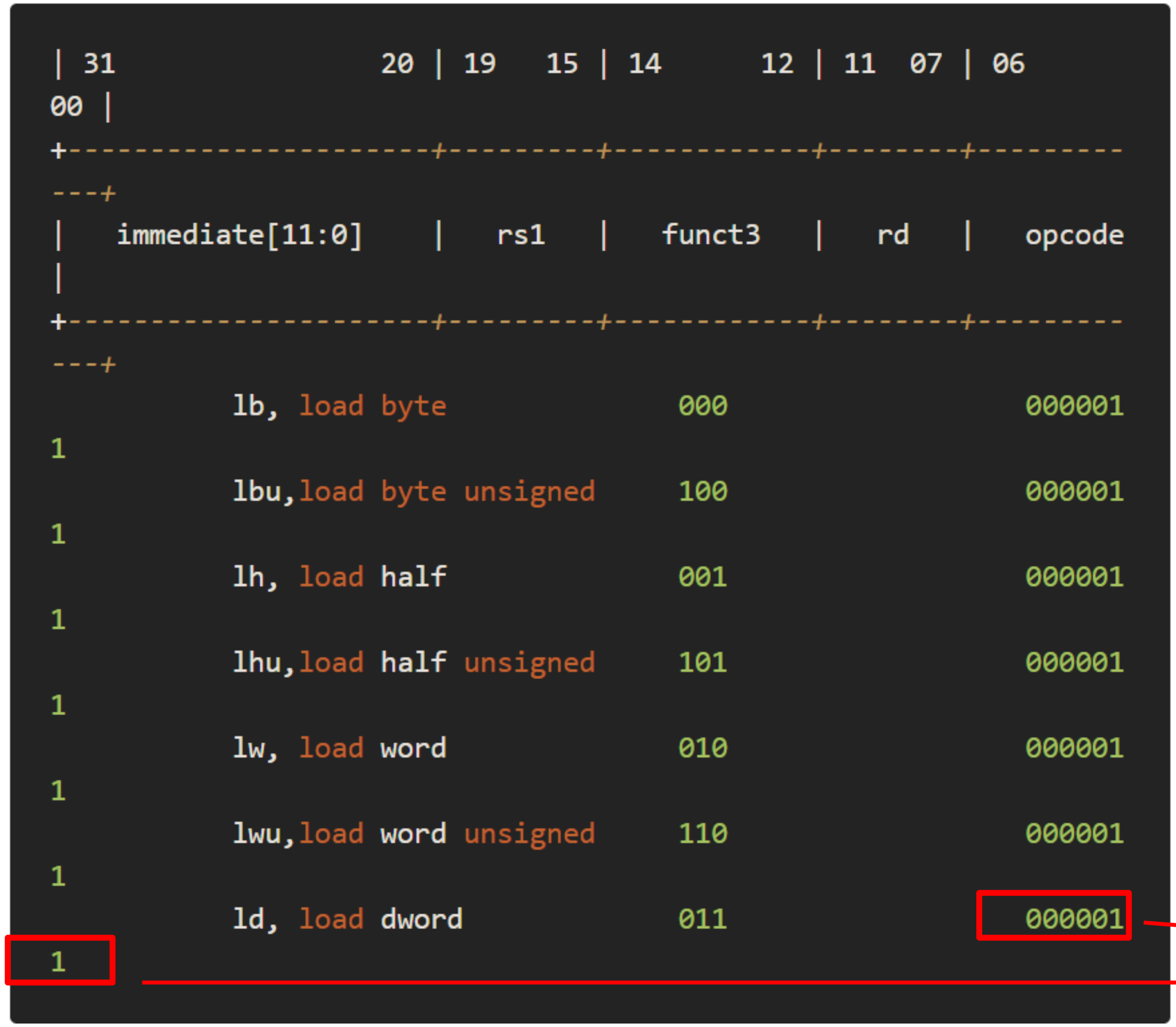
Register-Immediate ALU instructions.

Example:



Reg[x1] ← MEM[Reg[x2]+80]

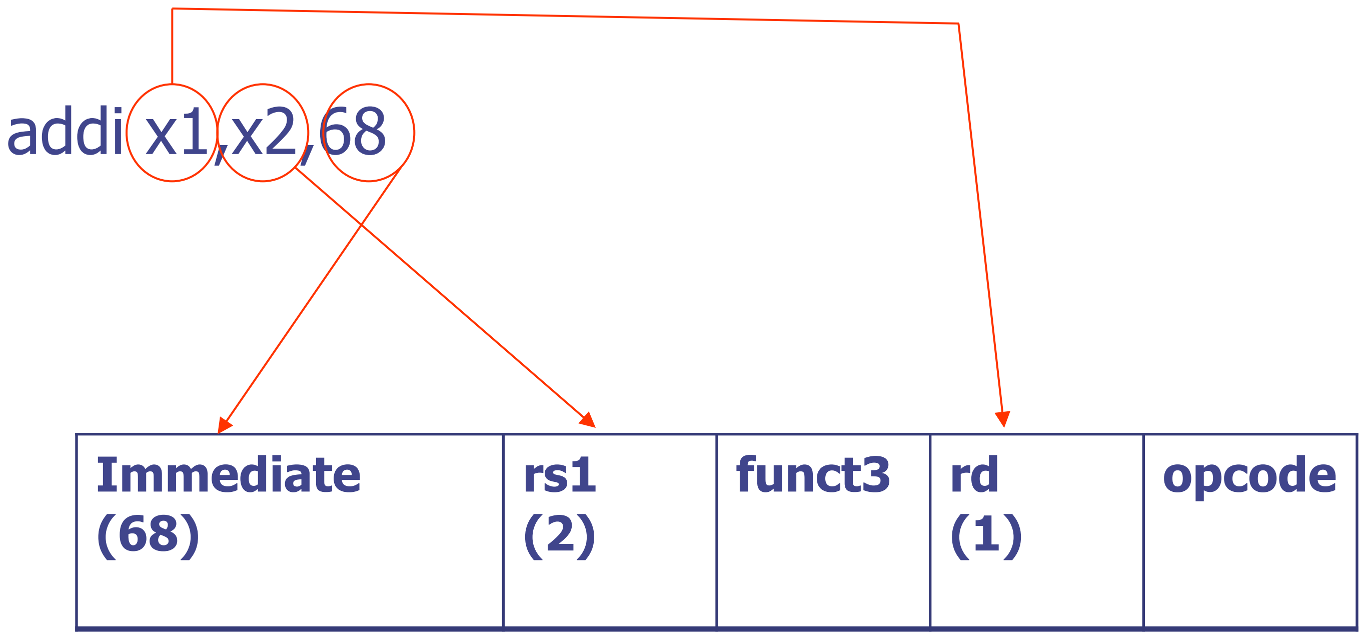
Load instructions such as **ld, lw, lh, lb** share the same opcode=0000011, but have different funct3 values.



```
| 31              20 | 19   15 | 14      12 | 11   07 | 06
00 |
+--------------------+---------+------------+---------+---------
---+
|   immediate[11:0]  |   rs1   |   funct3   |   rd    |   opcode
|
+--------------------+---------+------------+---------+---------
---+
           lb, load byte            000                   000001
1
           lbu,load byte unsigned   100                   000001
1
           lh, load half            001                   000001
1
           lhu,load half unsigned   101                   000001
1
           lw, load word            010                   000001
1
           lwu,load word unsigned   110                   000001
1
           ld, load dword           011                   000001
1
```

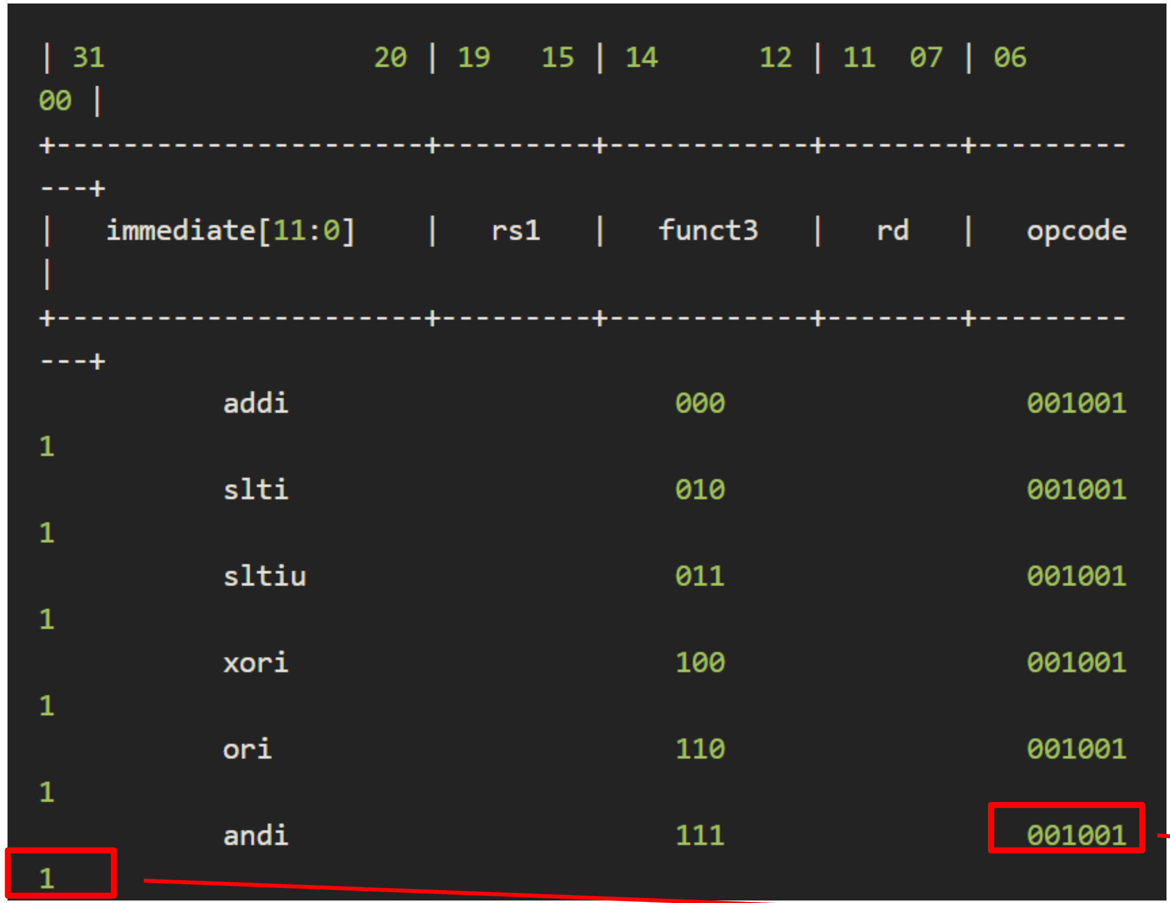
Opcode= 0000011

Example:



Reg[x1] ← Reg[x2]+68

ALU instructions such as **addi, slti, xori, ori, andi** share the same opcode=0010011, but have different funct3 values.



```
| 31              20 | 19    15 | 14       12 | 11   07 | 06
00 |
+--------------------+---------+------------+--------+---------
---+
|   immediate[11:0]    |   rs1   |   funct3   |   rd   |   opcode
|
+--------------------+---------+------------+--------+---------
---+
           addi                        000                    001001
1
           slti                        010                    001001
1
           sltiu                       011                    001001
1
           xori                        100                    001001
1
           ori                         110                    001001
1
           andi                        111                    001001
1
```

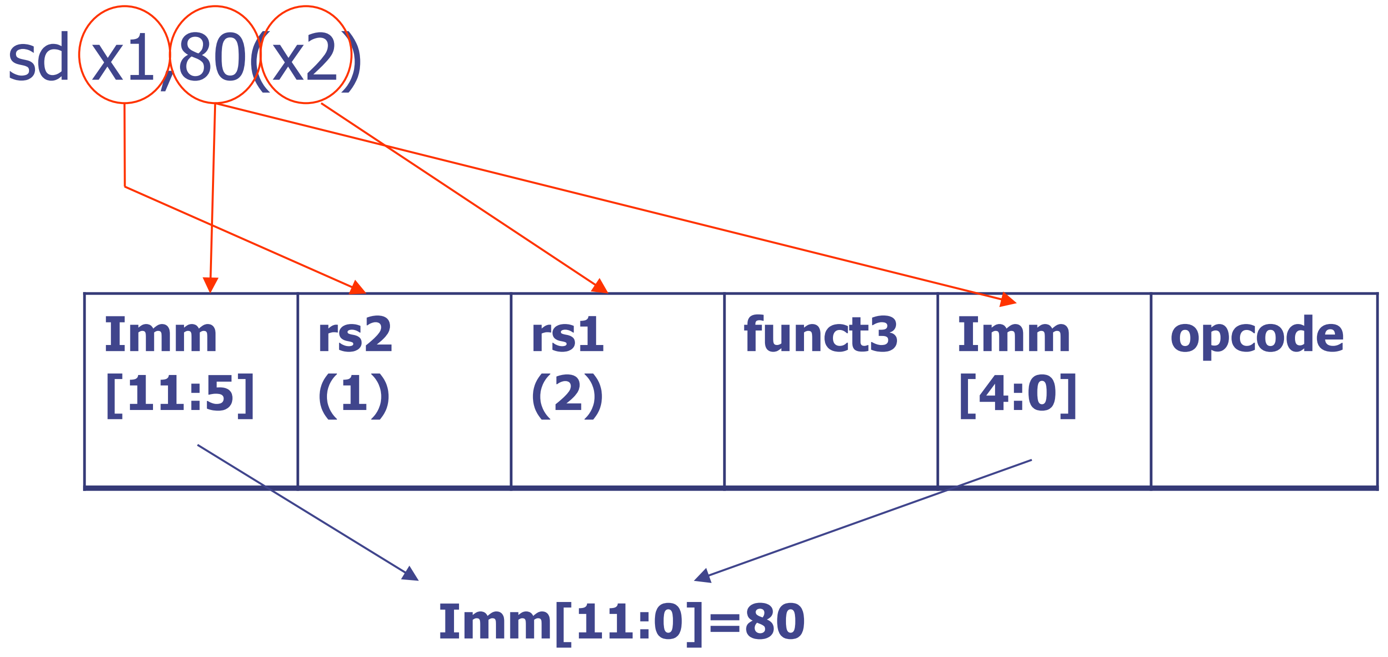
Opcode=
0010011

- S-type format

Imm [11:5]	rs2	rs1	funct3	Imm [4:0]	opcode
7	5	5	3	5	7

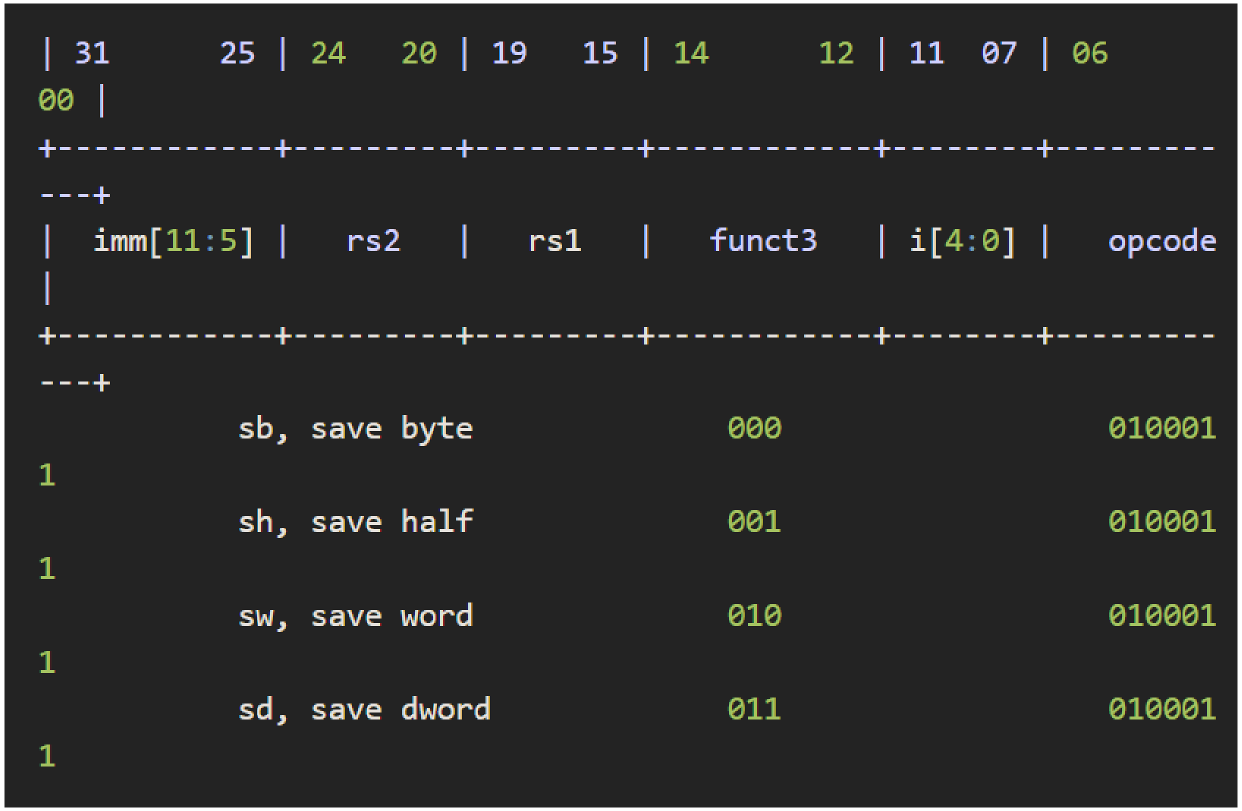
Store Instructions
Conditional Branch Instructions

Example:



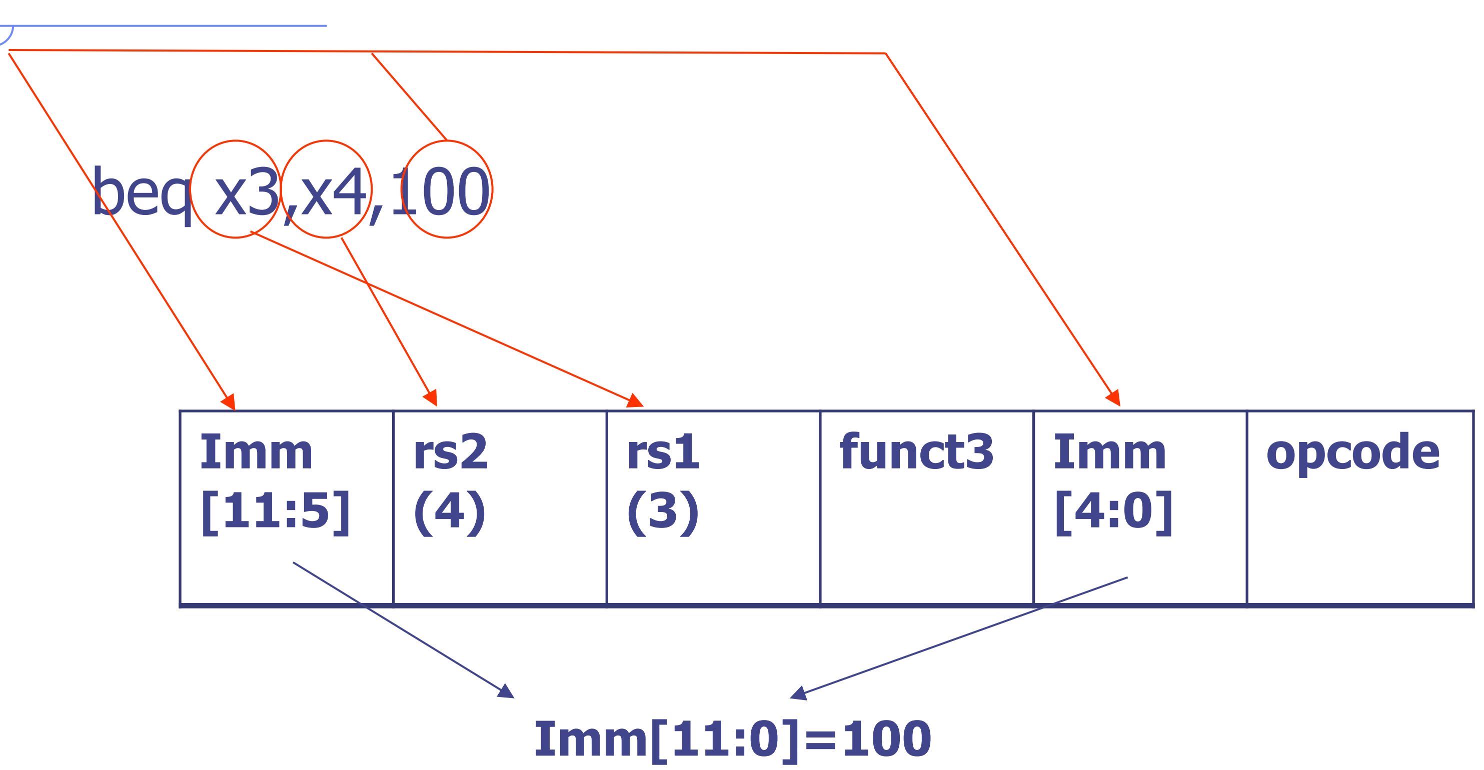
MEM[Reg[x2]+80] ← Reg[x1]

Store instructions such as **sb, sh, sw, sd** share the same opcode=0100011, but have different funct3 values.



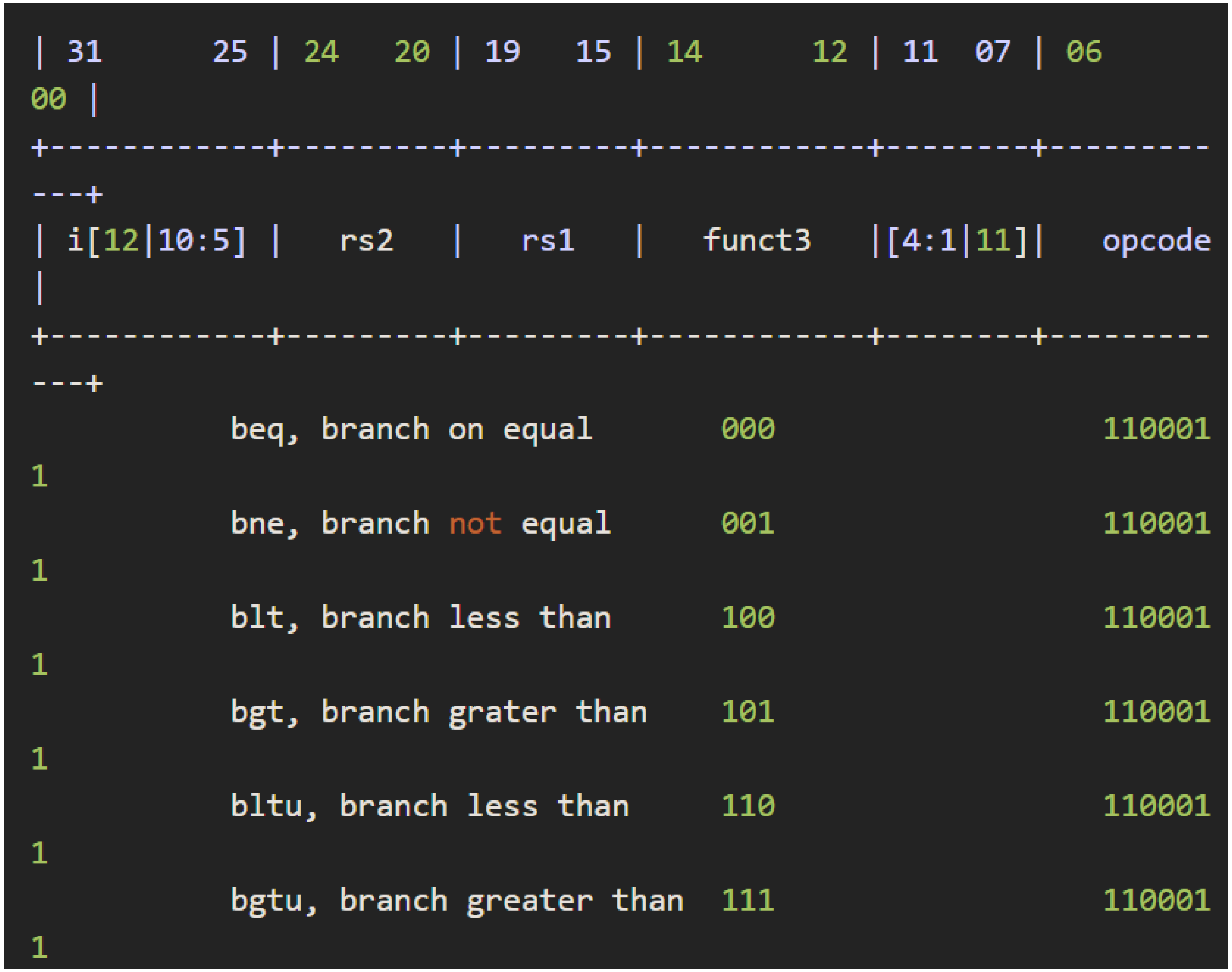
```
| 31      25 | 24   20 | 19   15 | 14      12 | 11  07 | 06
00 |
+------------+---------+---------+------------+--------+---------
---+
|  imm[11:5] |   rs2   |   rs1   |   funct3   | i[4:0] |   opcode
|
+------------+---------+---------+------------+--------+---------
---+
           sb, save byte           000                    010001
1
           sh, save half           001                    010001
1
           sw, save word           010                    010001
1
           sd, save dword          011                    010001
1
```

Example:



If(Reg[x3]==reg[x4]) PC ← PC +400

Branch instructions such as **beq, bne, blt, bgt, bltu, bgtu** share the same opcode=1100011, but have different funct3 values.



```
| 31       25 | 24    20 | 19    15 | 14        12 | 11   07 | 06
00 |
+------------+---------+---------+------------+--------+---------
---+
| i[12|10:5] |    rs2   |    rs1   |    funct3    |[4:1|11]|    opcode
|
+------------+---------+---------+------------+--------+---------
---+
            beq, branch on equal       000                    110001
1
            bne, branch not equal      001                    110001
1
            blt, branch less than      100                    110001
1
            bgt, branch grater than    101                    110001
1
            bltu, branch less than     110                    110001
1
            bgtu, branch greater than  111                    110001
1
```