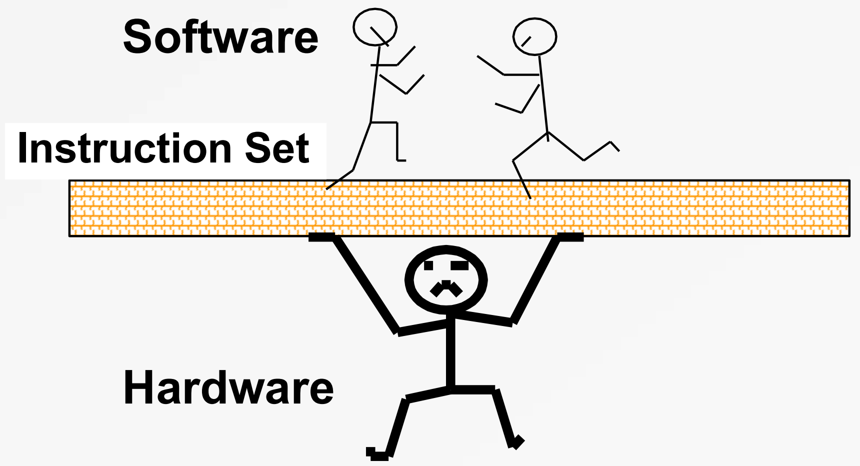
ISA
Instructure Set Archiceture

Instruction Set Architecture (ISA)



- The Instruction Set Architecture (ISA) is the part of the processor that is visible to the programmer or compiler writer.
- The ISA serves as the boundary between software and hardware.

Well Known ISAs

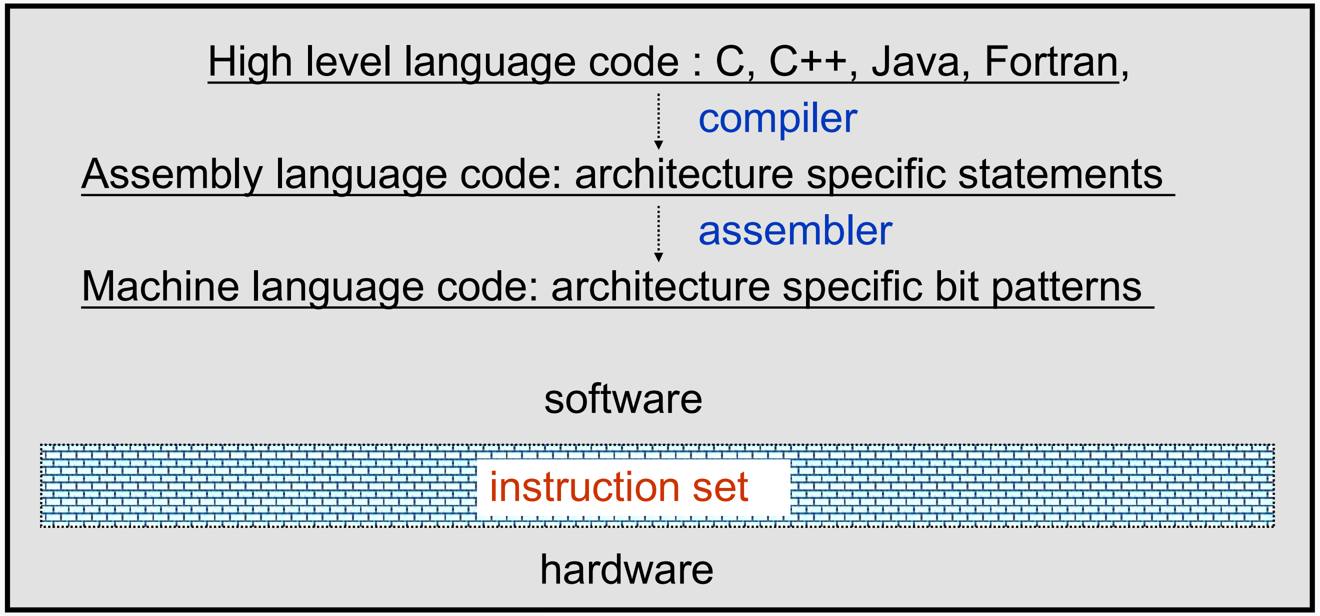
- x86
  - Based on Intel 8086 CPU in 1978
  - Intel family, also followed by AMD
  - X86-64
    - 64-bit extensions
    - Proposed by AMD, also followed by Intel
- ARM
  - 32-bit & 64-bit
  - Initially by Acorn RISC Machine
  - ARM Holding
- MIPS
  - 32-bit & 64-bit
  - By Microprocessor without Interlocked Pipeline Stages (MIPS) Technologies

Well Known ISAs

- SPARC
  - 32-bit & 64-bit
  - By Sun Microsystems
- PIC
  - 8-bit to 32-bit
  - By Microchip
- Z80
  - 8-bit
  - By Zilog in 1976
- Many extensions
  - Intel – MMX, SSE(streaming SIMD Extension, SSE2, AVX(Advanced vector extension)
  - AMD – 3D Now!

Instruction Set Architecture (ISA)

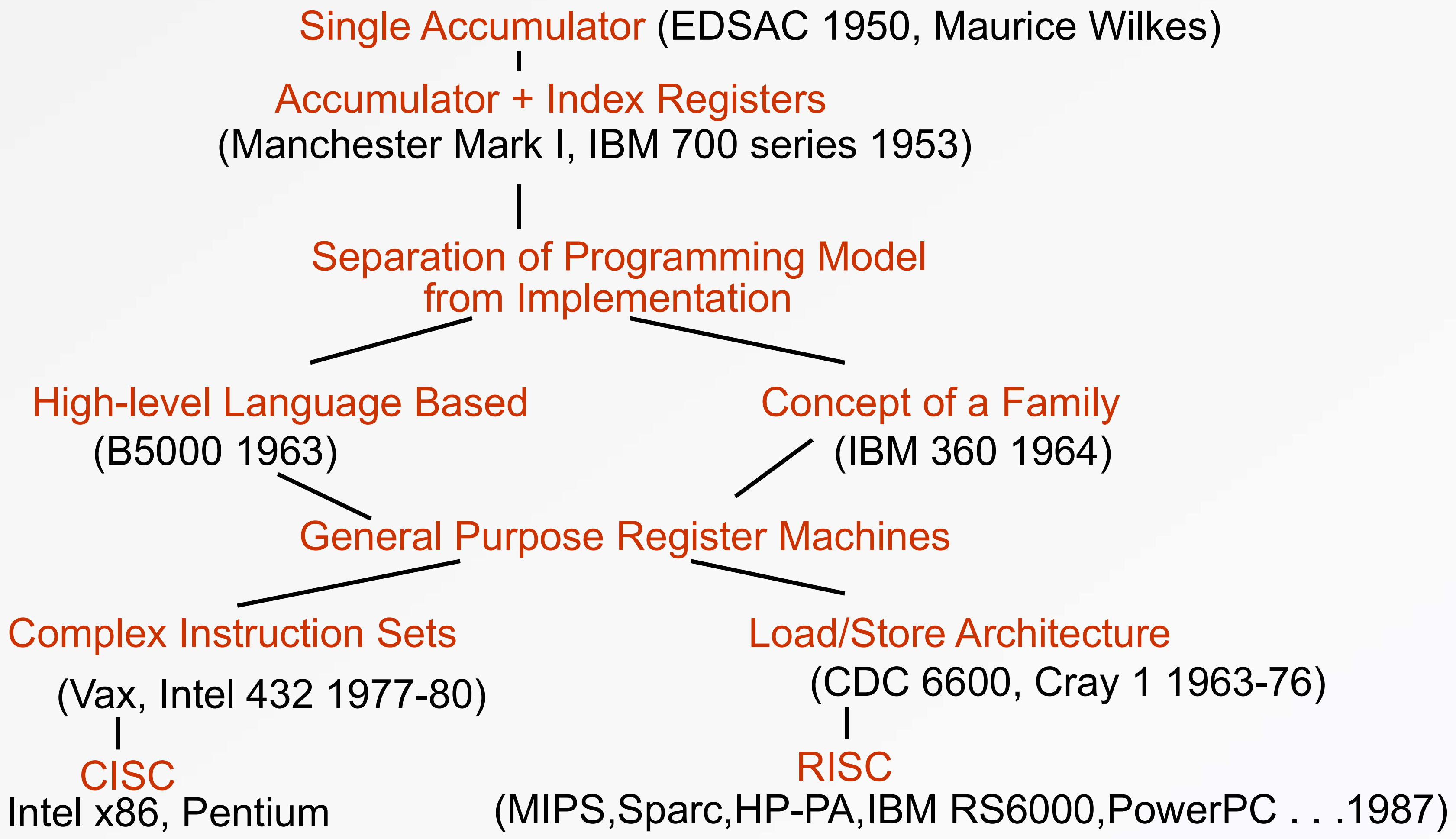
- Serves as an interface between software and hardware.
- Provides a mechanism by which the software tells the hardware what should be done.


High level language code : C, C++, Java, Fortran,
compiler
Assembly language code: architecture specific statements
assembler
Machine language code: architecture specific bit patterns
software
instruction set
hardware

Instruction Set Design Issues

- Instruction set design issues include:
  - Where are operands stored?
    - registers, memory, stack, accumulator
  - How many explicit operands are there?
    - 0, 1, 2, or 3
  - How is the operand location specified?
    - register, immediate, indirect, . . .
  - What type & size of operands are supported?
    - byte, int, float, double, string, vector. . .
  - What operations are supported?
    - add, sub, mul, move, compare . . .

Evolution of Instruction Sets



Single Accumulator (EDSAC 1950, Maurice Wilkes)

Accumulator + Index Registers
(Manchester Mark I, IBM 700 series 1953)

Separation of Programming Model
from Implementation

High-level Language Based
(B5000 1963)

Concept of a Family
(IBM 360 1964)

General Purpose Register Machines

Complex Instruction Sets
(Vax, Intel 432 1977-80)

CISC
Intel x86, Pentium

Load/Store Architecture
(CDC 6600, Cray 1 1963-76)

RISC
(MIPS,Sparc,HP-PA,IBM RS6000,PowerPC . . .1987)

Classifying ISAs

Accumulator (before 1960, e.g. 68HC11):

1-address	add A	acc ¬ acc + mem[A]

Stack (1960s to 1970s):

0-address	add	tos ¬ tos + next

Memory-Memory (1970s to 1980s):

2-address	add A, B	mem[A] ¬ mem[A] + mem[B]
3-address	add A, B, C	mem[A] ¬ mem[B] + mem[C]

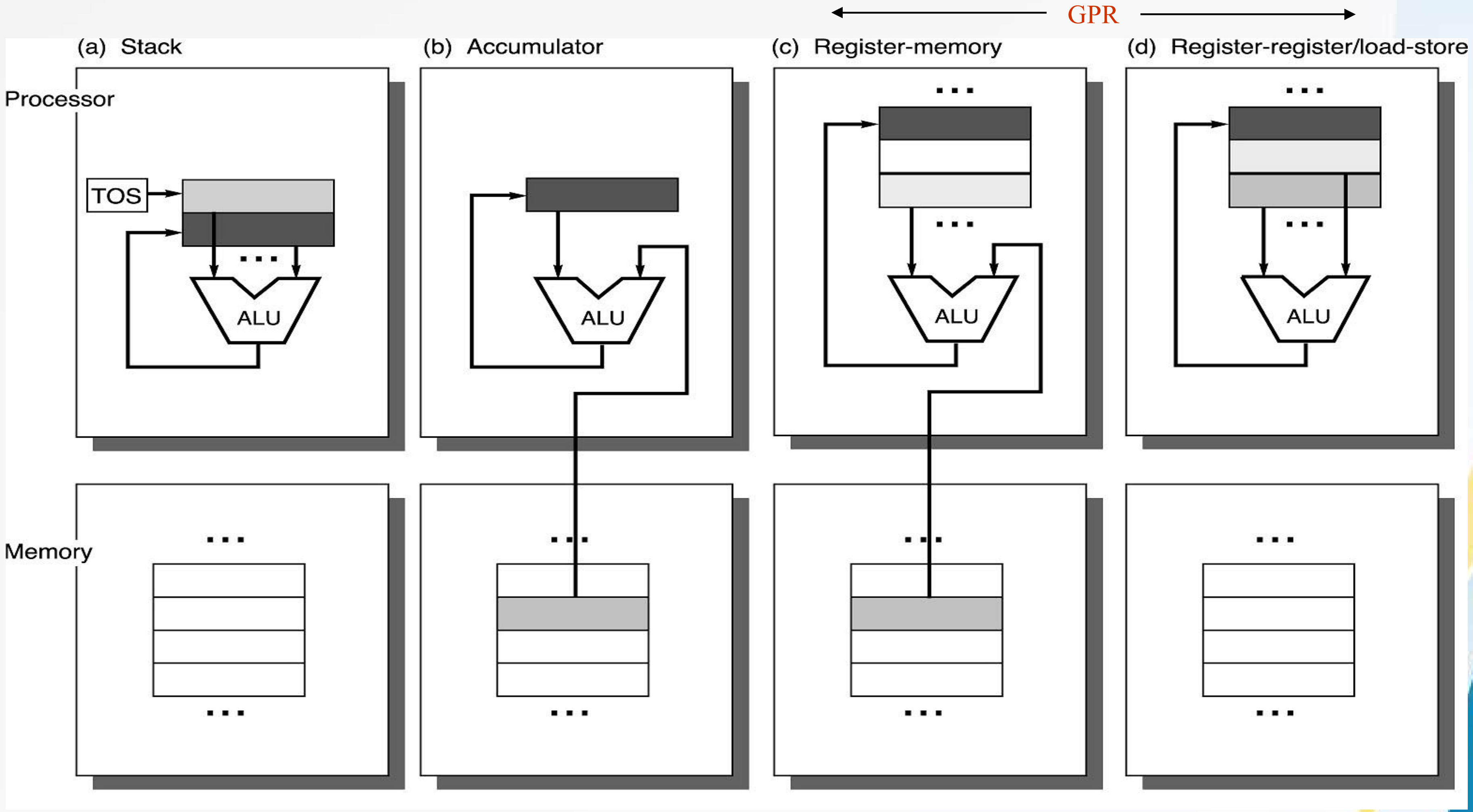
Register-Memory (1970s to present, e.g. 80x86):

2-address	add R1, A	R1 ¬ R1 + mem[A]
	load R1, A	R1 ¬ mem[A]

Register-Register (Load/Store) (1960s to present, e.g. MIPS):

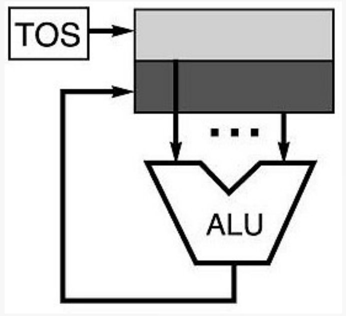
3-address	add R1, R2, R3	R1 ¬ R2 + R3
	load R1, R2	R1 ¬ mem[R2]
	store R1, R2	mem[R1] ¬ R2

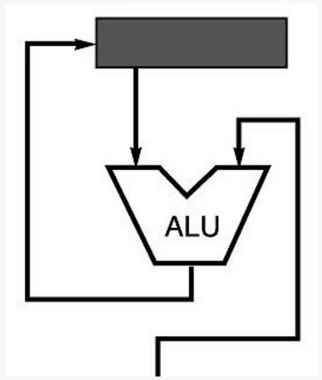
Operand Locations in Four ISA Classes

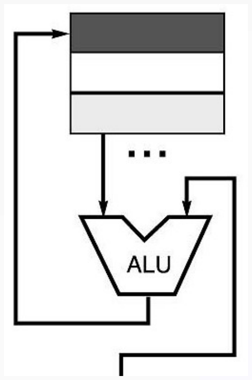


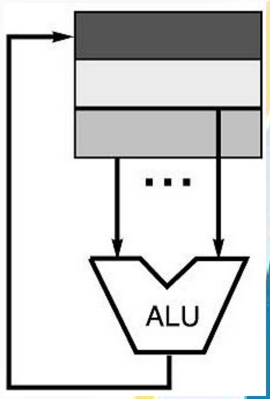
Code Sequence C = A + B for Four Instruction Sets

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



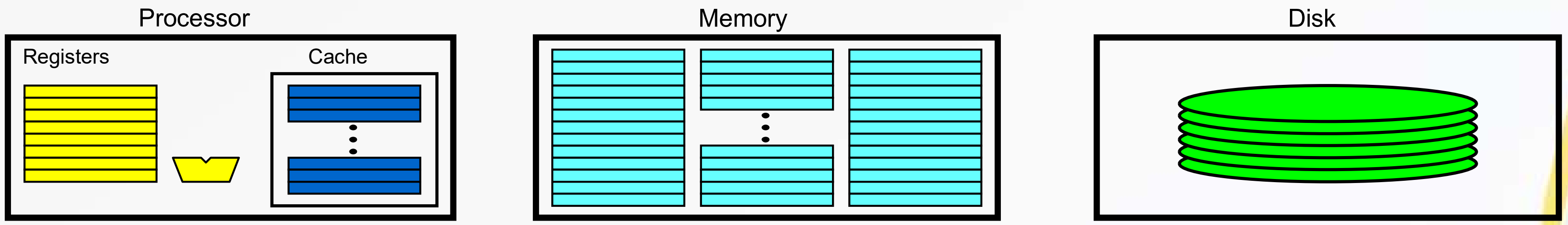




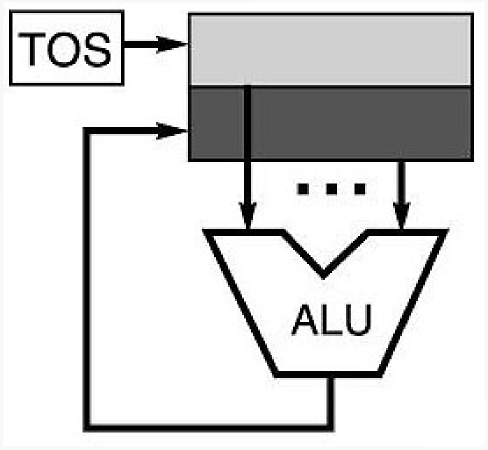


More About General Purpose Registers

- Why do almost all new architectures use GPRs?
  - Registers are much faster than memory (even cache)
    - Register values are available immediately
    - When memory isn't ready, processor must wait ("stall")
  - Registers are convenient for variable storage
    - Compiler assigns some variables just to registers
    - More compact code since small fields specify registers (compared to memory addresses)



Stack Architectures



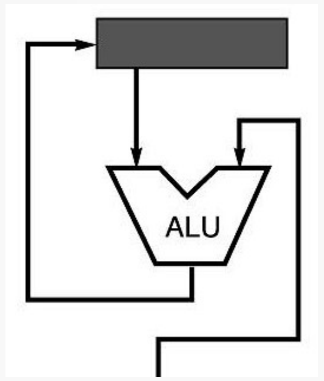
- Instruction set:
  add, sub, mult, div, . . .
  push A, pop A

- Example: A*B - (A+C*B)
  push A
  push B
  mul
  push A
  push C
  push B
  mul
  add
  sub

Accumulator Architectures

- Instruction set:
  - **add A, sub A, mult A, div A, . . .**
  - **load A, store A**

- Example: A*B - (A+C*B)
  - **load B**
  - **mul C**
  - **add A**
  - **store D**
  - **load A**
  - **mul B**
  - **sub D**



acc = acc +,-,*,/ mem[A]

Memory-Memory Architectures

- Instruction set:
  - **(3 operands) add A, B, C sub A, B, C mul A, B, C**
  - **(2 operands) add A, B sub A, B mul A, B**

- Example: A*B - (A+C*B)
  - **3 operands**

```
mul D, A, B
mul E, C, B
add E, A, E
sub E, D, E
```

2 operands

```
mov D, A
mul D, B
mov E, C
mul E, B
add E, A
sub E, D
```

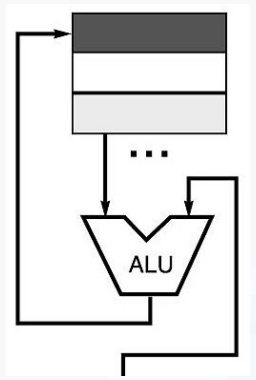
Register-Memory Architectures

- Instruction set:

```
add R1,  A          sub R1, A        mul R1, B
load R1, A          store R1, A
```

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

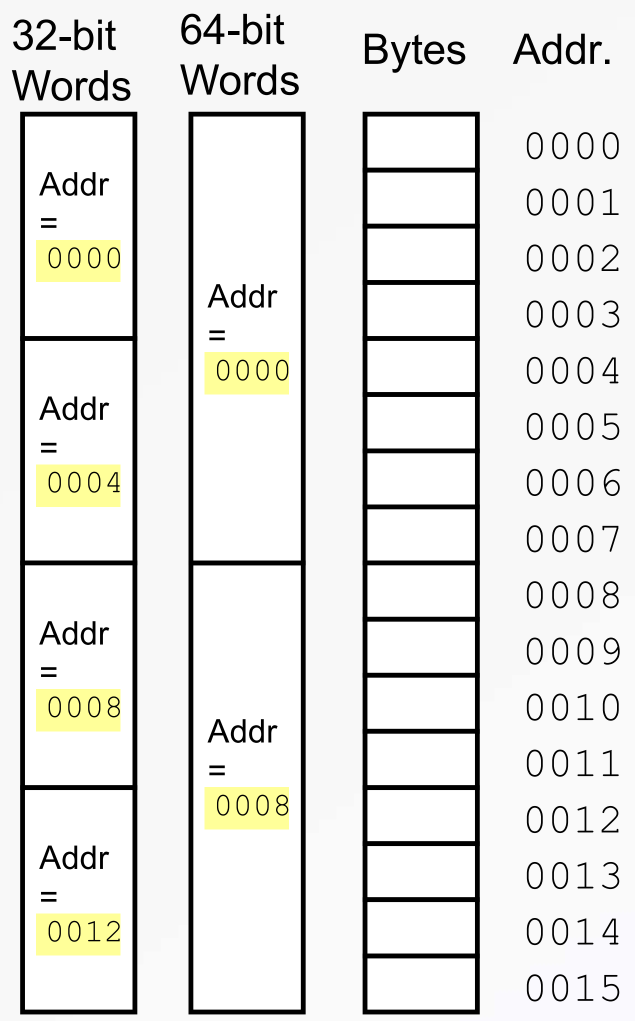
```
load R1, A
mul R1, B           /*        A*B                */
store R1, D
load R2, C
mul R2, B           /*        C*B                */
add R2, A           /*        A + CB             */
sub R2, D           /*        AB - (A + C*B)  */
```



R1 = R1 +,-,*,/ mem[B]

Word-Oriented Memory Organization

- Memory is byte addressed and provides access for bytes (8 bits), half words (16 bits), words (32 bits), and double words(64 bits).
- Addresses Specify Byte Locations
  - Address of first byte in word
  - Addresses of successive words differ by 4 (32-bit) or 8 (64-bit)

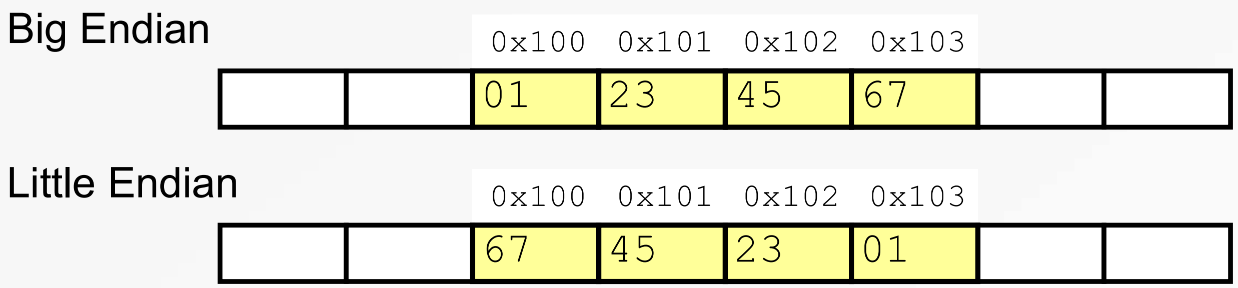


Byte Ordering

- How should bytes within multi-byte word be ordered in memory?
- Conventions
  - Sun’s, Mac’s are “Big Endian” machines
    - When lower byte addresses are used for the most significant byte(Left most byte) of the word.
  - Alphas, PC’s are “Little Endian” machines
    - When lower byte addresses are used for the less significant byte(Right most byte) of the word.

Byte Ordering Example

- Big Endian
  - Least significant byte has highest address
- Little Endian
  - Least significant byte has lowest address
- Example
  - Variable `x` has 4-byte representation `0x01234567`
  - Address given by `&x` is `0x100`



Instruction Set Architecture

Seven dimensions of an ISA

1. Class of ISA: Nearly all ISAs today are classified as general-purpose register architectures, the operands are either registers or memory locations.

- The 80x86 has 16 general-purpose and 16 registers
- MIPS has 32 general-purpose and 32 floating-point registers
- The two popular versions of this class are *register-memory* (80x86), which can access memory as part of many instructions,

and

- *load-store* (MIPS), which can access memory only with load or store instructions.
- All recent ISAs are load-store.

2. *Memory addressing*

- Virtually all desktop and server computers, including the 80x86 and MIPS, use byte addressing to access memory operands.
- MIPS, require that objects must be *aligned* .
- The 80x86 does not require alignment, but accesses are generally faster if operands are aligned.
- An access to an object of size *s* bytes at byte address *A* is aligned if $A \bmod s = 0$.

3. *Addressing modes*

- MIPS addressing modes are Register, Immediate (for constants), and Displacement, where a constant offset is added to a register to form the memory address.
- The 80x86 supports those three plus three variations of displacement: register indirect, indexed, and based with scaled index.

4. Types and sizes of operands

- Like most ISAs, MIPS and 80x86 support operand sizes of 8-bit (ASCII character), 16-bit (half word), 32-bit (integer or word), 64-bit (double word or long integer), and IEEE 754 floating point in 32-bit (single precision) and 64-bit (double precision).
- The 80x86 also supports 80-bit floating point (extended double precision).

5. Operations

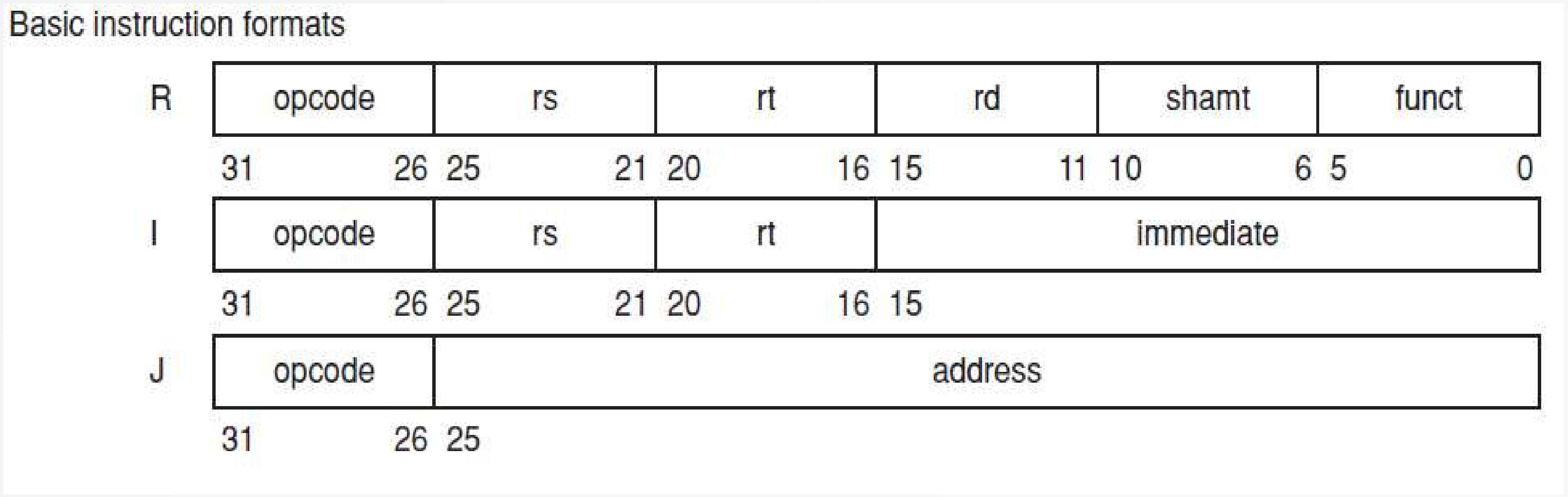
- The general categories of operations are data transfer, arithmetic logical, control and floating point.
- MIPS is a simple and easy-to-pipeline instruction set architecture, and it is representative of the RISC architectures.
- The 80x86 has a much richer and larger set of operations.

6. *Control flow instructions*

- Virtually all ISAs, including 80x86 and MIPS, support conditional branches, unconditional jumps, procedure calls, and returns.
- Both use PC-relative addressing, where the branch address is specified by an address field that is added to the PC.
- MIPS conditional branches (BE,BNE, etc.), while the 80x86 branches (JE, JNE, etc.)

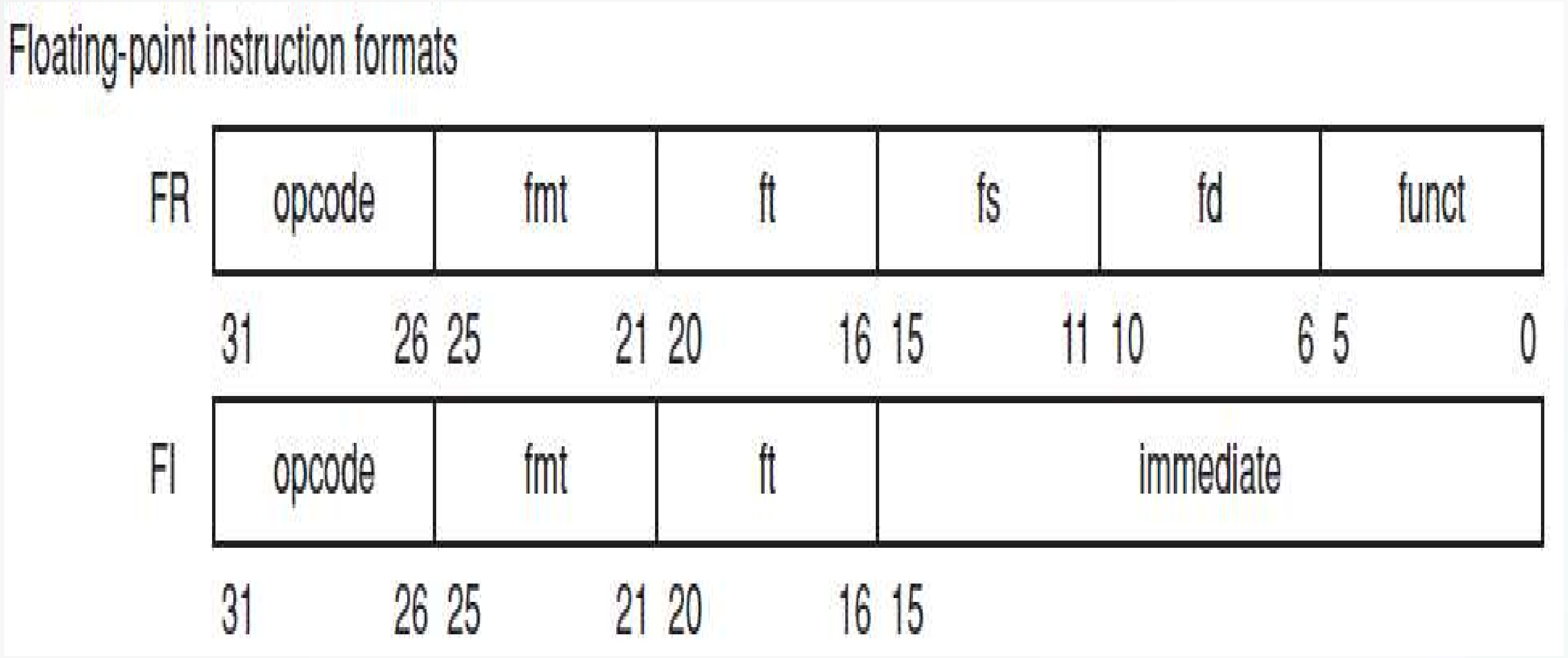
7. Encoding an ISA

- There are two basic choices on encoding: *fixed length* and *variable length* .
- All MIPS instructions are 32 bits long, which simplifies instruction decoding. Figure shows the MIPS instruction formats.
- The 80x86 encoding is variable length, ranging from 1 to 18 bytes. Variable length instructions can take less space than fixed-length instructions.



- opcode (6 bits): Operation code
- rs (5 bits): first source operand register
- rt (5 bits): second source operand register
- rd (5 bits): destination operand register
- shamt (5 bits): shift amount for logical operation
- funct (6 bits): function code selects the specific variant of opcode

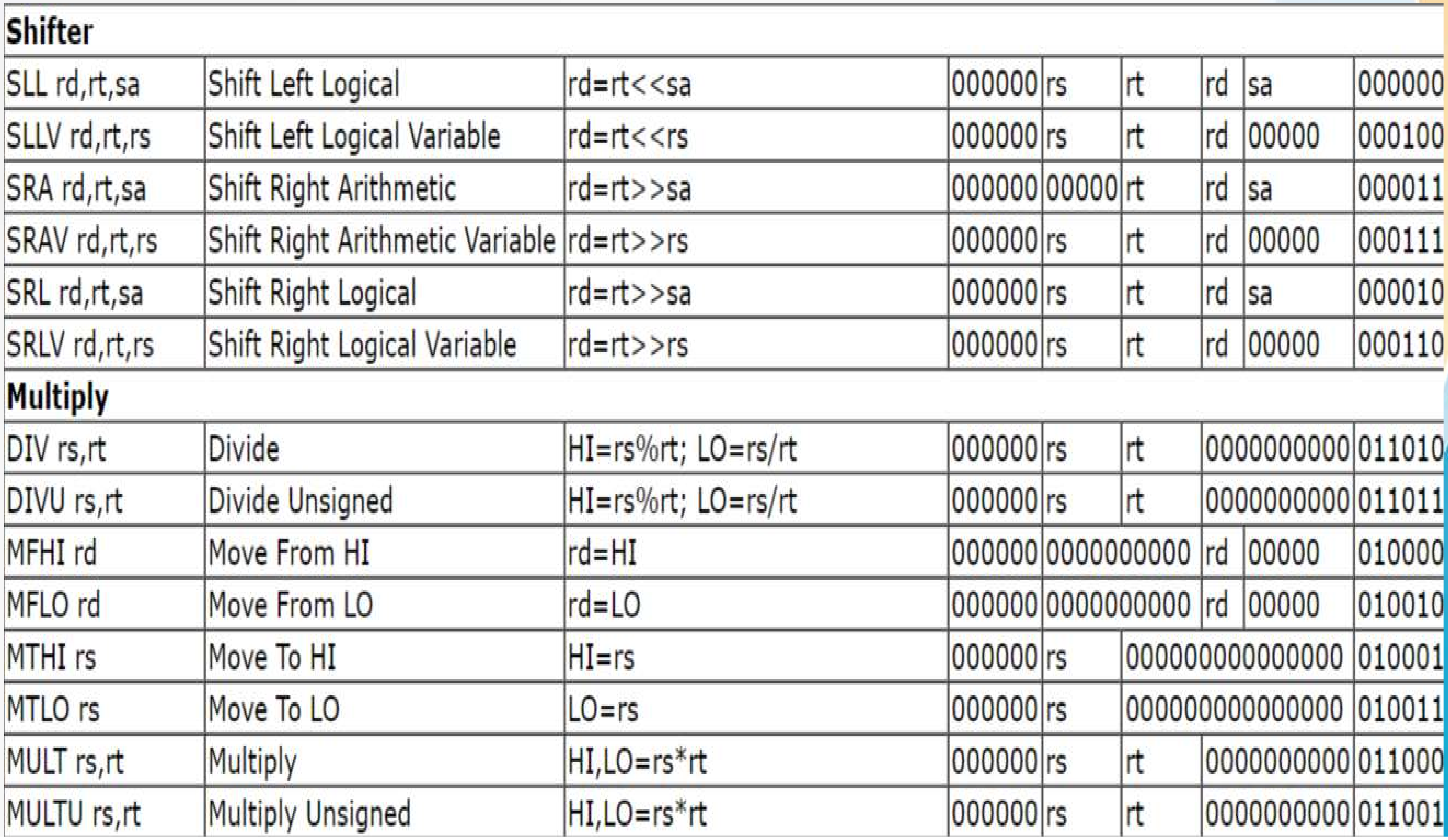
- Immidiate (16 bits): constant after immidiate instruction
- address (26 bits): offset value after immidiate instruction



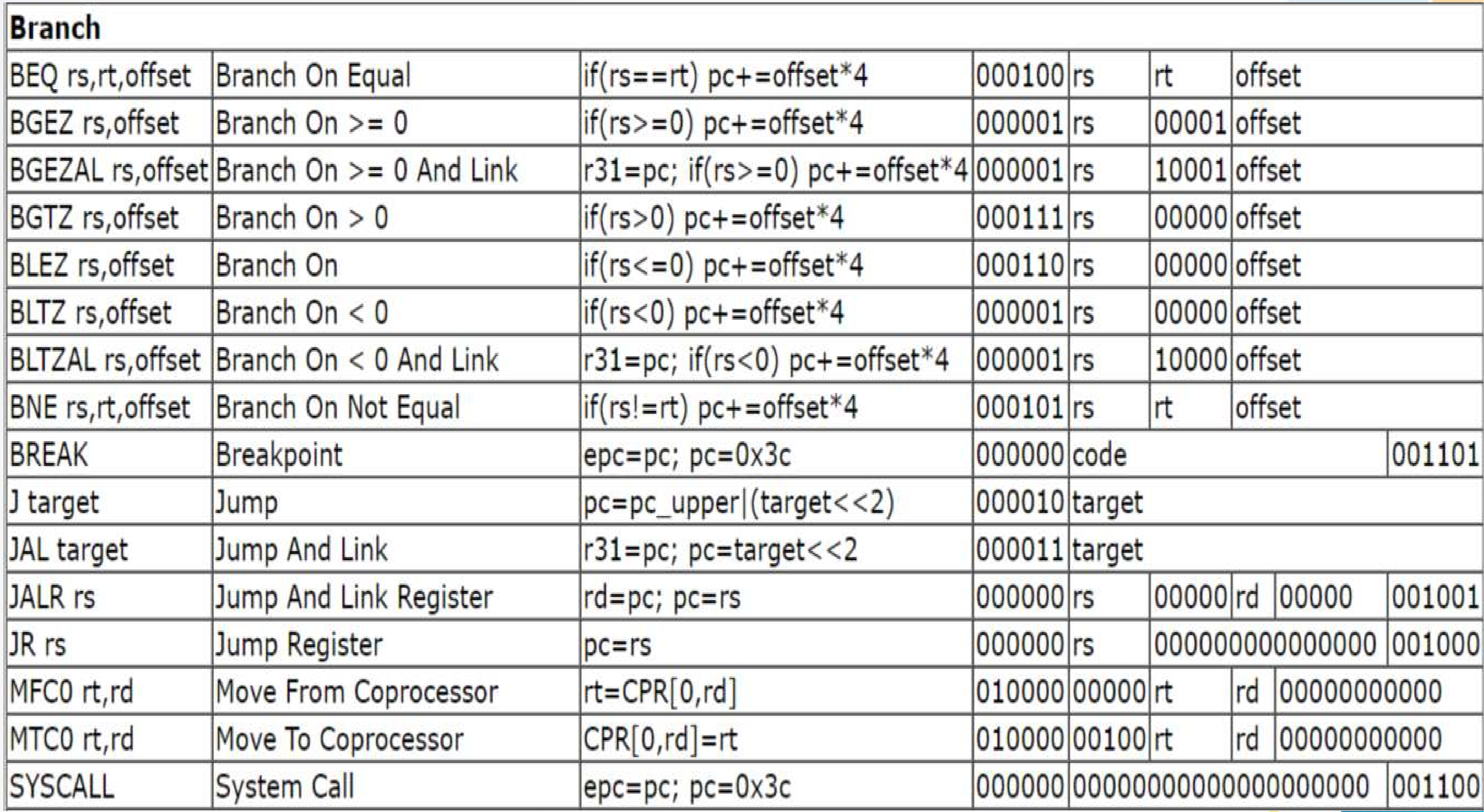
FR format for floating point

operations, and the FI format for floating point branches.

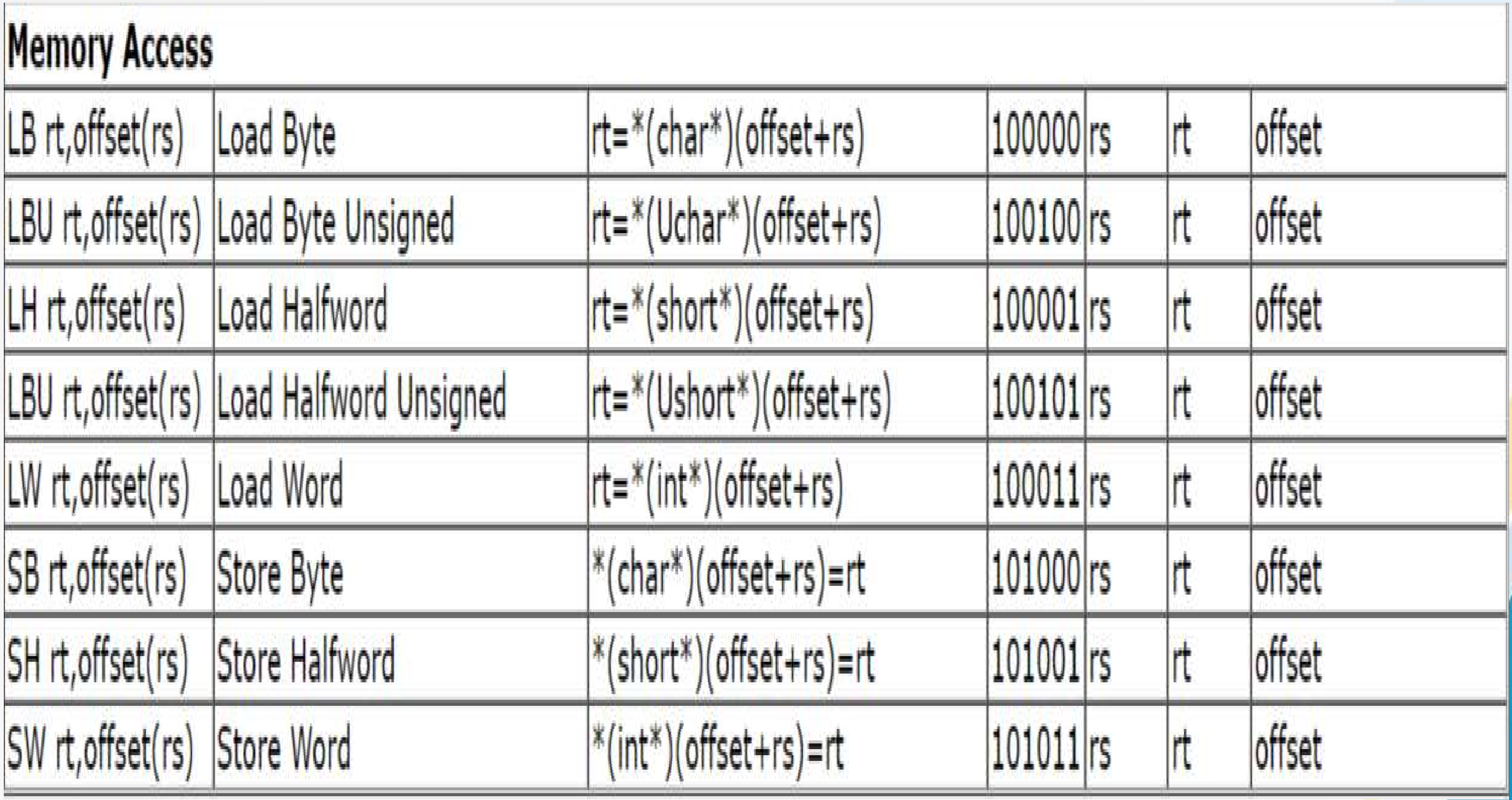
Arithmetic Logic Unit								
ADD rd,rs,rt	Add	rd=rs+rt	000000	rs	rt	rd	00000	100000
ADDI rt,rs,imm	Add Immediate	rt=rs+imm	001000	rs	rt	imm		
ADDIU rt,rs,imm	Add Immediate Unsigned	rt=rs+imm	001001	rs	rt	imm		
ADDU rd,rs,rt	Add Unsigned	rd=rs+rt	000000	rs	rt	rd	00000	100001
AND rd,rs,rt	And	rd=rs&rt	000000	rs	rt	rd	00000	100100
ANDI rt,rs,imm	And Immediate	rt=rs&imm	001100	rs	rt	imm		
LUI rt,imm	Load Upper Immediate	rt=imm<<16	001111	rs	rt	imm		
NOR rd,rs,rt	Nor	rd=~(rs|rt)	000000	rs	rt	rd	00000	100111
OR rd,rs,rt	Or	rd=rs|rt	000000	rs	rt	rd	00000	100101
ORI rt,rs,imm	Or Immediate	rt=rs|imm	001101	rs	rt	imm		
SLT rd,rs,rt	Set On Less Than	rd=rs<rt	000000	rs	rt	rd	00000	101010
SLTI rt,rs,imm	Set On Less Than Immediate	rt=rs<imm	001010	rs	rt	imm		
SLTIU rt,rs,imm	Set On < Immediate Unsigned	rt=rs<imm	001011	rs	rt	imm		
SLTU rd,rs,rt	Set On Less Than Unsigned	rd=rs<rt	000000	rs	rt	rd	00000	101011
SUB rd,rs,rt	Subtract	rd=rs-rt	000000	rs	rt	rd	00000	100010
SUBU rd,rs,rt	Subtract Unsigned	rd=rs-rt	000000	rs	rt	rd	00000	100011
XOR rd,rs,rt	Exclusive Or	rd=rs^rt	000000	rs	rt	rd	00000	100110
XORI rt,rs,imm	Exclusive Or Immediate	rt=rs^imm	001110	rs	rt	imm		



Shifter								
SLL rd,rt,sa	Shift Left Logical	rd=rt<<sa	000000	rs	rt	rd	sa	000000
SLLV rd,rt,rs	Shift Left Logical Variable	rd=rt<<rs	000000	rs	rt	rd	00000	000100
SRA rd,rt,sa	Shift Right Arithmetic	rd=rt>>sa	000000	00000	rt	rd	sa	000011
SRAV rd,rt,rs	Shift Right Arithmetic Variable	rd=rt>>rs	000000	rs	rt	rd	00000	000111
SRL rd,rt,sa	Shift Right Logical	rd=rt>>sa	000000	rs	rt	rd	sa	000010
SRLV rd,rt,rs	Shift Right Logical Variable	rd=rt>>rs	000000	rs	rt	rd	00000	000110
Multiply								
DIV rs,rt	Divide	HI=rs%rt; LO=rs/rt	000000	rs	rt	0000000000		011010
DIVU rs,rt	Divide Unsigned	HI=rs%rt; LO=rs/rt	000000	rs	rt	0000000000		011011
MFHI rd	Move From HI	rd=HI	000000	0000000000		rd	00000	010000
MFLO rd	Move From LO	rd=LO	000000	0000000000		rd	00000	010010
MTHI rs	Move To HI	HI=rs	000000	rs	000000000000000			010001
MTLO rs	Move To LO	LO=rs	000000	rs	000000000000000			010011
MULT rs,rt	Multiply	HI,LO=rs*rt	000000	rs	rt	0000000000		011000
MULTU rs,rt	Multiply Unsigned	HI,LO=rs*rt	000000	rs	rt	0000000000		011001



Branch								
BEQ rs,rt,offset	Branch On Equal	if(rs==rt) pc+=offset*4	000100	rs	rt	offset		
BGEZ rs,offset	Branch On >= 0	if(rs>=0) pc+=offset*4	000001	rs	00001	offset		
BGEZAL rs,offset	Branch On >= 0 And Link	r31=pc; if(rs>=0) pc+=offset*4	000001	rs	10001	offset		
BGTZ rs,offset	Branch On > 0	if(rs>0) pc+=offset*4	000111	rs	00000	offset		
BLEZ rs,offset	Branch On	if(rs<=0) pc+=offset*4	000110	rs	00000	offset		
BLTZ rs,offset	Branch On < 0	if(rs<0) pc+=offset*4	000001	rs	00000	offset		
BLTZAL rs,offset	Branch On < 0 And Link	r31=pc; if(rs<0) pc+=offset*4	000001	rs	10000	offset		
BNE rs,rt,offset	Branch On Not Equal	if(rs!=rt) pc+=offset*4	000101	rs	rt	offset		
BREAK	Breakpoint	epc=pc; pc=0x3c	000000	code				001101
J target	Jump	pc=pc_upper|(target<<2)	000010	target				
JAL target	Jump And Link	r31=pc; pc=target<<2	000011	target				
JALR rs	Jump And Link Register	rd=pc; pc=rs	000000	rs	00000	rd	00000	001001
JR rs	Jump Register	pc=rs	000000	rs	000000000000000			001000
MFC0 rt,rd	Move From Coprocessor	rt=CPR[0,rd]	010000	00000	rt	rd	00000000000	
MTC0 rt,rd	Move To Coprocessor	CPR[0,rd]=rt	010000	00100	rt	rd	00000000000	
SYSCALL	System Call	epc=pc; pc=0x3c	000000	00000000000000000000				001100



Memory Access

LB rt,offset(rs)	Load Byte	rt=*(char*)(offset+rs)	100000	rs	rt	offset
LBU rt,offset(rs)	Load Byte Unsigned	rt=*(Uchar*)(offset+rs)	100100	rs	rt	offset
LH rt,offset(rs)	Load Halfword	rt=*(short*)(offset+rs)	100001	rs	rt	offset
LBU rt,offset(rs)	Load Halfword Unsigned	rt=*(Ushort*)(offset+rs)	100101	rs	rt	offset
LW rt,offset(rs)	Load Word	rt=*(int*)(offset+rs)	100011	rs	rt	offset
SB rt,offset(rs)	Store Byte	*(char*)(offset+rs)=rt	101000	rs	rt	offset
SH rt,offset(rs)	Store Halfword	*(short*)(offset+rs)=rt	101001	rs	rt	offset
SW rt,offset(rs)	Store Word	*(int*)(offset+rs)=rt	101011	rs	rt	offset