Assembly Language

Readings: 2.1-2.7, 2.9-2.10, 2.14

Green reference card

Assembly language

- Simple, regular instructions – building blocks of C, Java & other languages
- Typically one-to-one mapping to machine language

Our goal

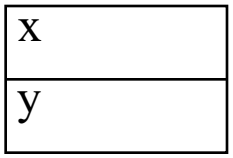
- Understand the basics of assembly language
- Help figure out what the processor needs to be able to do

Not our goal to teach complete assembly/machine language programming

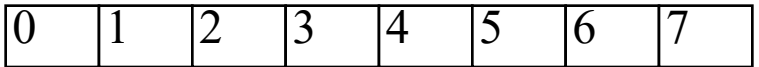
- Floating point
- Procedure calls
- Stacks & local variables

Aside: C/C++ Primer

```
struct coord { int x, y; }; /* Declares a type                    */
struct coord start;         /* Object with two slots, x and y     */
start.x = 1;                /* For objects "." accesses a slot    */
struct coord *myLoc;        /* "*" is a pointer to objects        */
myLoc = &start;             /* "&" returns thing's location       */
myLoc->y = 2;               /* "->" is "*" plus "."               */
```




```
int scores[8];              /* 8 ints, from 0..7 */
scores[1]=5;                /* Access locations in array */
int *index = scores;        /* Points to scores[0] */
index++;                    /* Next scores location */
(*index)++;                 /* "*" works in arays as well */
index = &(scores[3]);       /* Points to scores[3] */
*index = 9;
```



ARM Assembly Language

The basic instructions have four components:

- Operator name
- Destination
- 1st operand
- 2nd operand

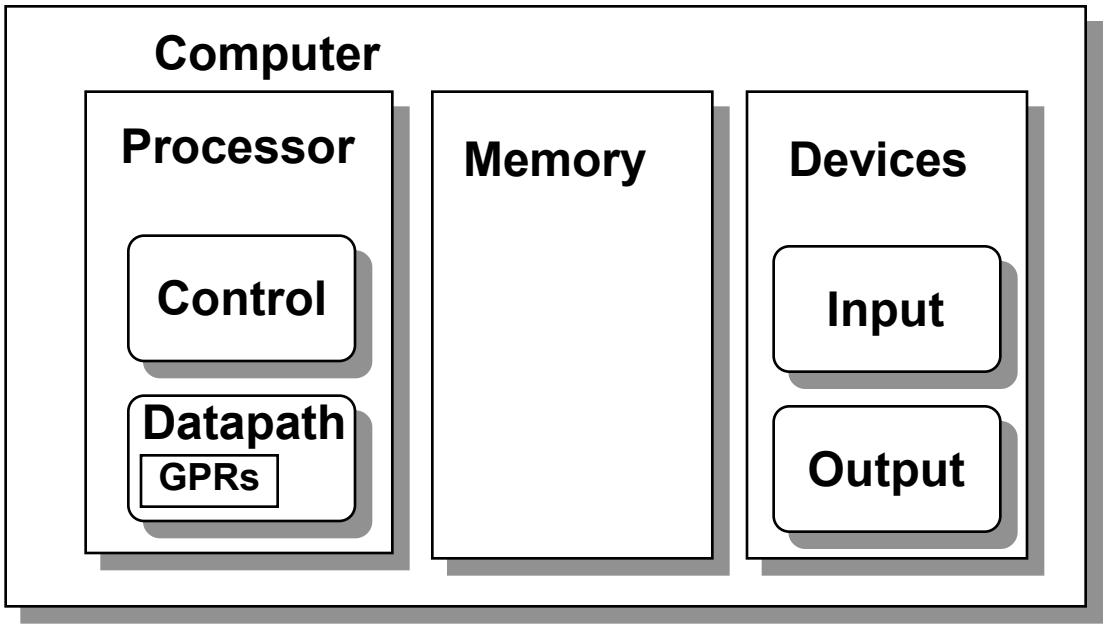
```
ADD <dst>, <src1>, <src2>      // <dst> = <src1> + <src2>
SUB <dst>, <src1>, <src2>      // <dst> = <src1> - <src2>
```

Simple format: easy to implement in hardware

More complex: `A = B + C + D – E`

Operands & Storage

For speed, CPU has 32 general-purpose registers for storing most operands
For capacity, computer has large memory (multi-GB)



Load/store operation moves information between registers and main memory
All other operations work on registers

Registers

32x 64-bit registers for operands

Register	Function	Comment
X0-X7	Function arguments/Results	
X8	Result, if a pointer	
X9-X15	Volatile Temporaries	Not saved on call
X16-X17	Linker scratch registers	Don’t use them
X18	Platform register	Don’t use this
X19-X27	Temporaries (saved across calls)	Saved on call
X28	Stack Pointer	
X29	Frame Pointer	
X30	Return Address	
X31	Always 0	No-op on write

Basic Operations

(Note: just subset of all instructions)

Mathematic: ADD, SUB, MUL, SDIV

```
ADD X0, X1, X2     // X0 = X1+X2
```

Immediate (second input a constant)

```
ADDI X0, X1, #100  // X0 = X1+100
```

Logical: AND, ORR, EOR

```
AND X0, X1, X2     // X0 = X1&X2
```

Immediate

```
ANDI X0, X1, #7    // X0 = X1&b0111
```

Shift: left & right logical (LSL, LSR)

```
LSL X0, X1, #4     // X0 = X1<<4
```

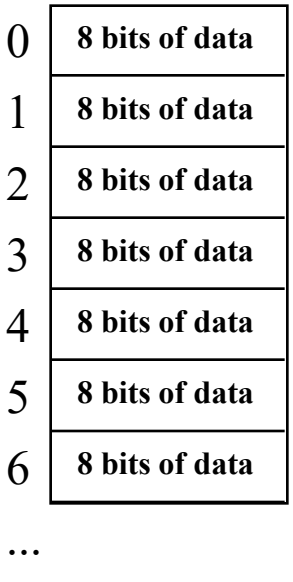
Example: Take bits 6-4 of X0 and make them bits 2-0 of X1, zeros otherwise:

Memory Organization

Viewed as a large, single-dimension array, with an address.

A memory address is an index into the array

"Byte addressing" means that the index points to a byte of memory.

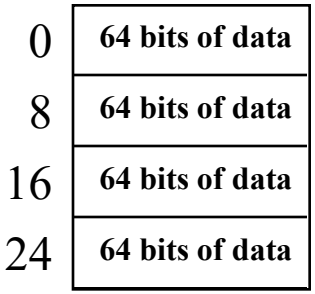


Memory Organization (cont.)

Bytes are nice, but most data items use larger units.

Double-word = 64 bits = 8 bytes

Word = 32 bits = 4 bytes



Registers hold 64 bits of data

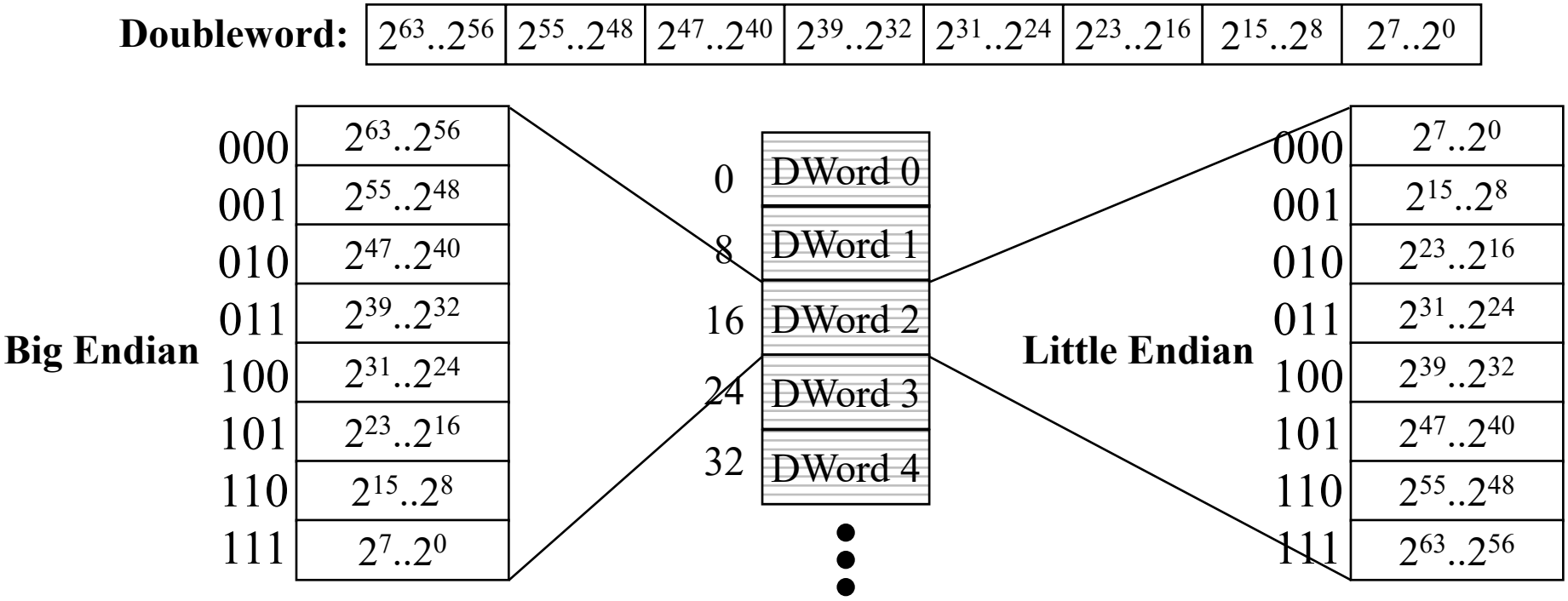
2^{64} bytes with byte addresses from 0 to 2^{64}-1

2^{61} double-words with byte addresses 0, 8, 16, ... 2^{64}-8

Double-words and words are aligned

i.e., what are the least 3 significant bits of a double-word address?

Addressing Objects: Endian and Alignment



Big Endian: address of most significant byte = doubleword address
Motorola 68k, MIPS, IBM 360/370, Xilinx Microblaze, Sparc

Little Endian: address of least significant byte = doubleword address
Intel x86, DEC Vax, Altera Nios II, Z80

ARM: can do either – this class assumes Little-Endian.

Data Storage

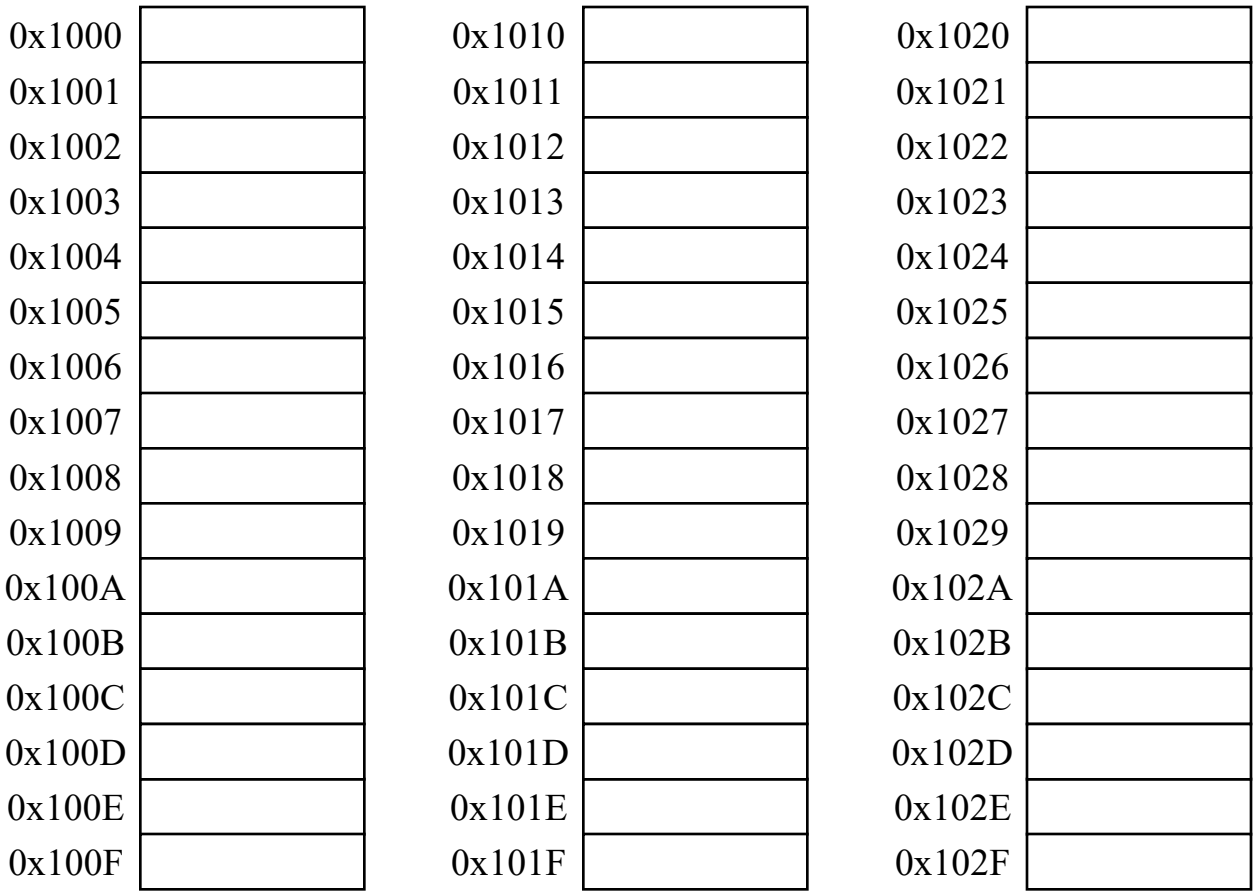
Characters: 8 bits (byte)

Integers: 64 bits (D-word)

Array: Sequence of locations

Pointer: Address (64 bits)

```
// G = ASCII 71: 0x47
char a = 'G';
int x = 258; // 0x102
char *b;
int *y;
b = new char[4];
y = new int[10];
```



(Note: real compilers place local variables (the "stack") from beginng of memory, new'ed structures (the "heap") from end. We ignore that here for simplicity)

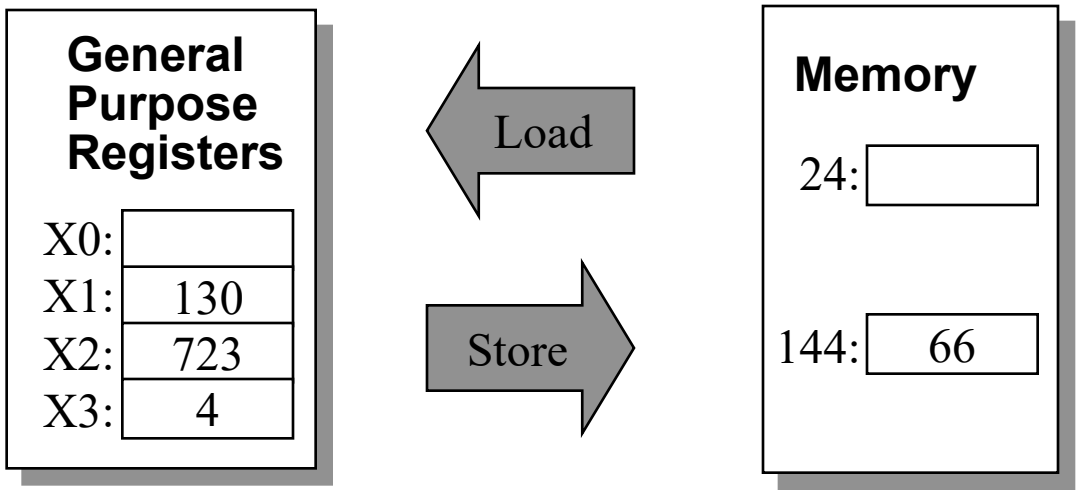
Loads & Stores

Loads & Stores move data between memory and registers

All operations on registers, but too small to hold all data

```
LDUR X0, [X1, #14]        // X0 = Memory[X1+14]

STUR X2, [X3, #20]        // Memory[X3+20] = X2
```



Note: LDURB & STURB load & store bytes

Addressing Example

The address of the start of a character array is stored in X0. Write assembly to load the following characters

X2 = Array[0]

X3 = Array[1]

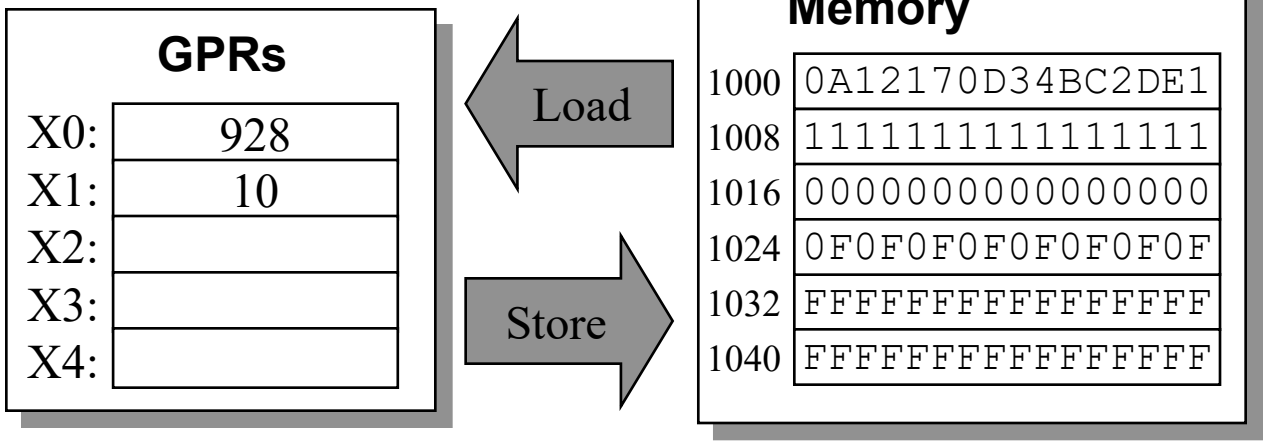
X4 = Array[2]

X5 = Array[k] // Assume the value of k is in X1

Array Example

```
/* Swap the kth and (k+1)th element of an array */
swap(int v[], int k) {
      int temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
}

// Assume v in X0, k in X1
```

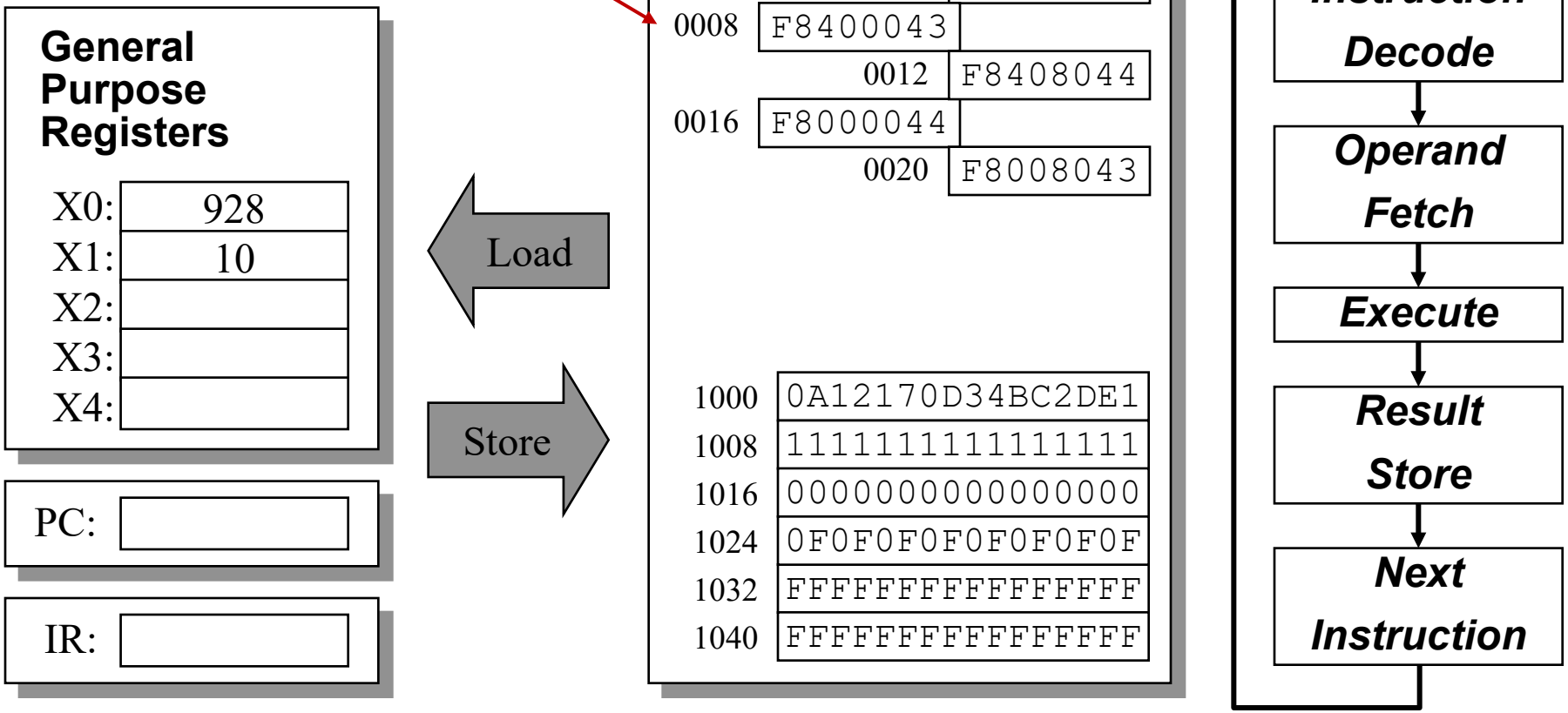


Execution Cycle Example

PC: Program Counter

IR: Instruction Register

Note:
Word addresses
Instructions are 32b



Flags/Condition Codes

Flag register holds information about result of recent math operation

- Negative: was result a negative number?
- Zero: was result 0?
- Overflow: was result magnitude too big to fit into 64-bit register?
- Carry: was the carry-out true?

Operations that set the flag register contents:

- ADDS, ADDIS, ANDS, ANDIS, SUBS, SUBIS, some floating point.

Most commonly used are subtracts, so we have a synonym: CMP

- CMP X0, X1 same as SUBS X31, X0, X1
- CMPI X0, #15 same as SUBIS X31, X0, #15

Control Flow

Unconditional Branch – GOTO different next instruction

```
B START              // go to instruction labeled with "START" label
BR X30               // go to address in X30: PC = value of X30
```

Conditional Branches – GOTO different next instruction if condition is true

1 register: CBZ (==0), CBNZ (!= 0)

```
CBZ X0, FOO      // if X0 == 0 GOTO FOO: PC = Address of instr w/FOO label
```

2 register: B.LT (<), B.LE(<=), B.GE (>=), B.GT(>), B.EQ(==), B.NE(!=)

first compare (CMP X0, X1, CMPI X0, #12), then b.cond instruction

```
CMP X0, X1       // compare X0 with X1 - same as SUBS X31, X0, X1
B.EQ FOO         // if X0 == X1 GOTO FOO: PC = Address of instr w/FOO label
```

```
if (a == b)
   a = a + 3;
else
   b = b + 7;
c = a + b;
```

```
// X0 = a, X1 = b, X2 = c
     CMP X0, X1          // set flags
     B.NE ELSEIF         // branch if a!=b
     ADDI X0, X0, #3     // a = a + 3
     B DONE              // avoid else
ELSEIF:
     ADDI X1, X1, #7     // b = b + 7
DONE:
     ADD X2, X0, X1      // c = a + b
```

Loop Example

Compute the sum of the values 0…N-1

```
int sum = 0;
for (int I = 0; I != N; I++) {
        sum += I;
}
// X0 = N, X1 = sum, X2 = I
```

String toUpper

Convert a string to all upper case

```
    char *index = string;
    while (*index != 0) { /* C strings end in 0 */
        if (*index >= 'a' && *index <= 'z')
            *index = *index +('A' - 'a');
        index++;
    }
// string is a pointer held at Memory[80].
// X0=index, 'A' = 65, 'a' = 97, 'z' = 122
```

Machine Language vs. Assembly Language

Assembly Language

- mnemonics for easy reading
- labels instead of fixed addresses
- Easier for programmers
- Almost 1-to-1 with machine language

Machine language

- Completely numeric representation
- format CPU actually uses

```
SWAP:
    LSL     X9, X1, #3                  11010011011 00000 000011 00001 01001
    ADD     X9, X0, X9      // Compute address of v[k]   10001011000 01001 000000 00000 01001
    LDUR    X10, [X9, #0]   // get v[k]                  11111000010 000000000 00 01001 01010
    LDUR    X11, [X9, #8]   // get v[k+1]                11111000010 000001000 00 01001 01011
    STUR    X11, [X9, #0]   // save new value to v[k]    11111000000 000000000 00 01001 01011
    STUR    X10, [X9, #8]   // save new value to v[k+1]  11111000000 000001000 00 01001 01010
    BR      X30             // return from subroutine    11010110000 00000 000000 00000 11110
```

Labels

Labels specify the address of the corresponding instruction

- Programmer doesn't have to count line numbers
- Insertion of instructions doesn't require changing entire code

```
// X0 = N, X1 = sum, X2 = I
   ADD X1, X31, X31   // sum = 0
   ADD X2, X31, X31   // I = 0
TOP:
   CMP X2, X0         // Check I vs N
   B.GE END           // end when !(I<N)
   ADD X1, X1, X2     // sum += I
   ADDI X2, X2, #1    // I++
   B TOP              // next iteration
END:
```

Notes:

- Branches are PC-relative

  PC = PC + 4*(BranchOffset)

- BranchOffset positive -> branch downward. Negative -> branch upward.

Labels Example

Compute the value of the labels in the code below.

Branches: PC = PC + 4*(BranchOffset)

```
// Program starts at address 100
    LDUR   X0, [X31, #100]
LOOP:
    LDURB  X1, [X0, #0]
    CBZ    X1, END
    CMPI   X1, #97
    B.LT   NEXT
    CMPI   X1, #122
    B.GT   NEXT
    SUBI   X1, X1, #32
    STURB  X1, [X0, #0]
NEXT:
    ADDI   X0, X0, 1
    B      LOOP
END:
```

Instruction Types

Can group instructions by # of operands

3-register

```
ADD X0, X1, X2
ADDI X0, X1, #100
```

2-register

```
AND X0, X1, X2
ANDI X0, X1, #7
LSL X0, X1, #4
LSR X0, X1, #2
```

1-register

```
LDUR X0, [X1, #14]
LDURB X0, [X1, #14]
STUR X0, [X1, #14]
STURB X0, [X1, #14]
B START
BR X30
```

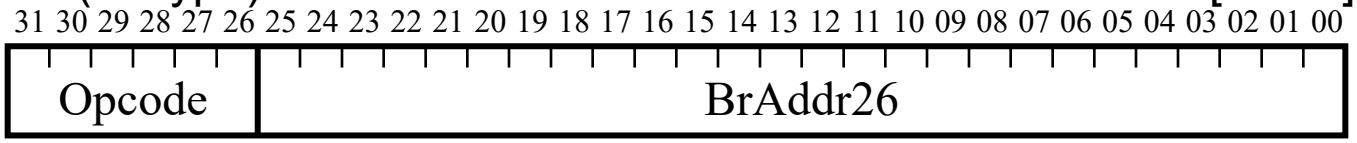
0-register

```
CBZ X0, FOO
B.EQ DEST
```

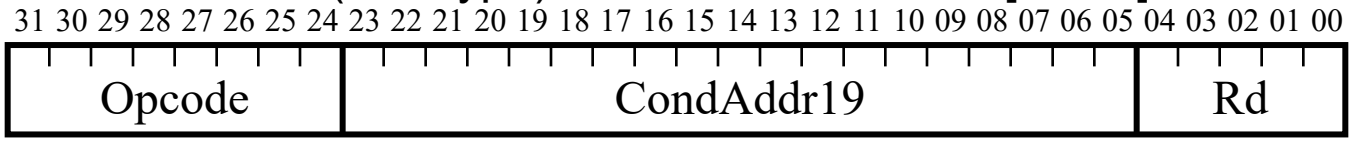
Instruction Formats

All instructions encoded in 32 bits (operation + operands/immediates)

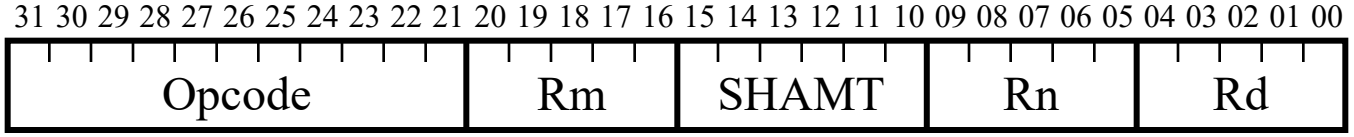
Branch (B-Type) Instr[31:21] = 0A0-0BF



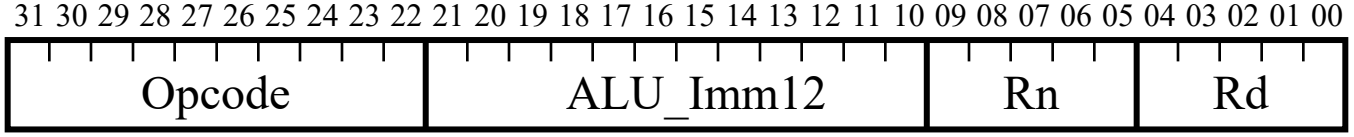
Conditional Branch (CB-Type) Instr[31:21] = 2A0-2A7, 5A0-5AF



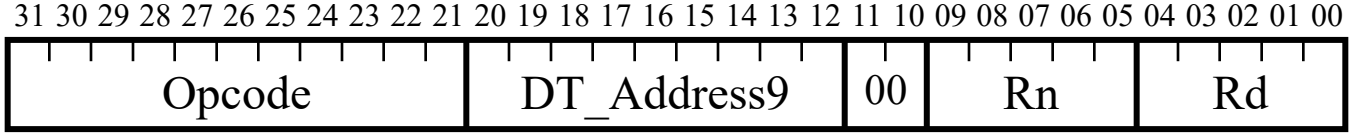
Register (R-Type) Instr[31:21] = 450-458, 4D6-558, 650-658, 69A-758



Immediate (I-Type) Instr[31:21] = 488-491, 588-591, 688-691, 788-791

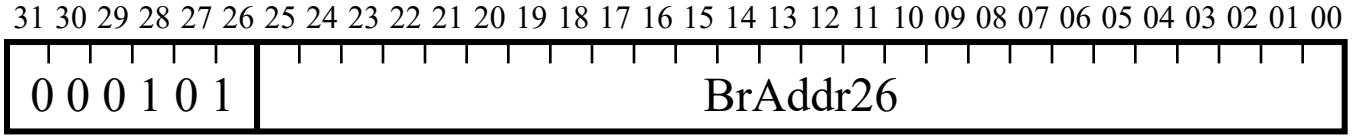


Memory (D-Type) Instr[31:21] = 1C0-1C2, 7C0-7C2



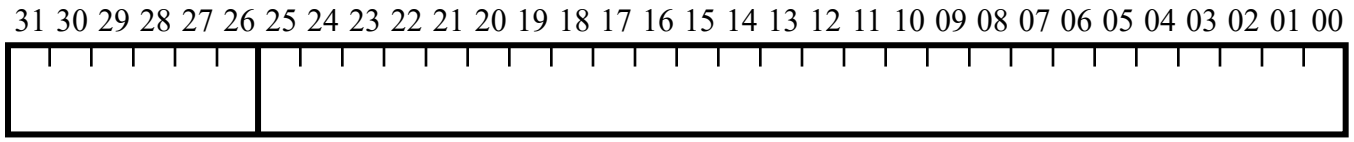
B-Type

Used for unconditional branches



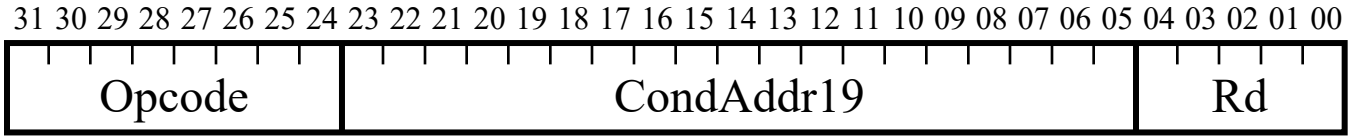
0x05: B

```
B -3    // PC = PC + 4*-3
```



CB-Type

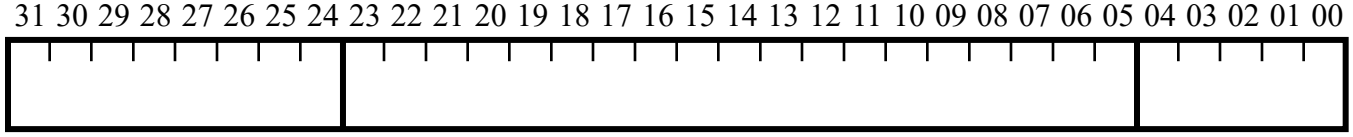
Used for conditional branches



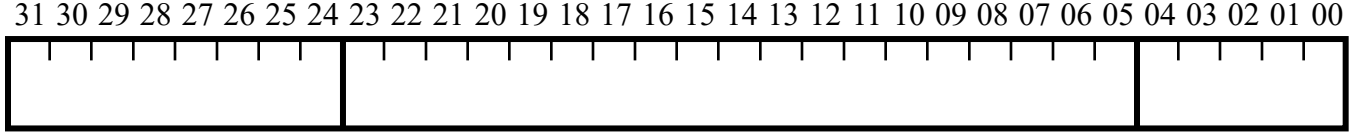
0x54: B.cond
0xB4: CBZ
0xB5: CBNZ

Condition Codes
0x00: EQ (==)
0x01: NE (!=)
0x0A: GE (>=)
0x0B: LT (<)
0x0C: GT (>)
0x0D: LE (<=)

```
CBZ X12, -3        // if(X12==0) PC = PC + 4*-3
```

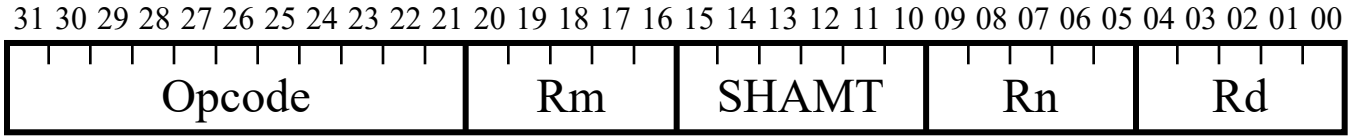



```
B.LT -5            // if (lessThan) PC = PC + 4*-5
```



R-Type

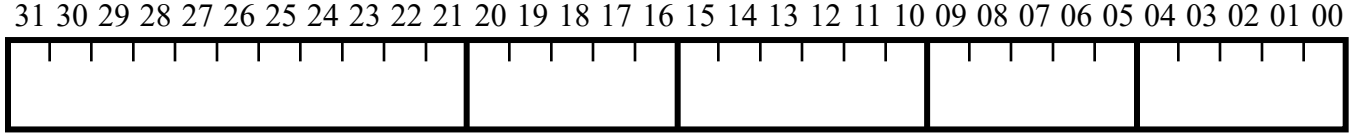
Used for 3 register ALU operations and shift



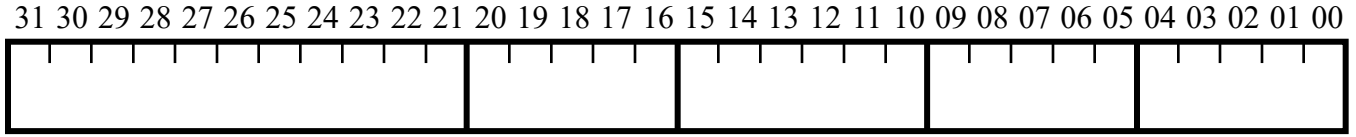
0x450: AND
0x458: ADD
0x4D6: SDIV, shamt=02
0x4D8: MUL, shamt=1F
0x550: ORR
0x558: ADDS
0x650: EOR
0x658: SUB
0x69A: LSR
0x69B: LSL
0x6B0: BR, rest all 0's but Rd
0x750: ANDS
0x758: SUBS

Op2 (0 for shift) | Shift amount (0 for non-shift) | Op1 | Dest

```
ADD       X3, X5, X6          // X3 = X5+X6
```

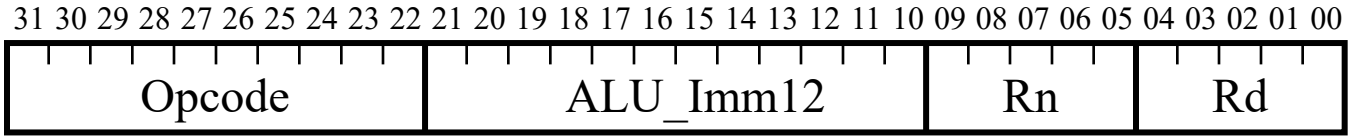



```
LSL       X10, X4, #6         // X10 = X4<<6
```



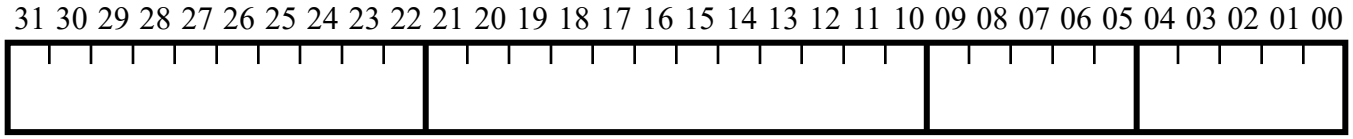
I-Type

Used for 2 register & 1 constant ALU operations



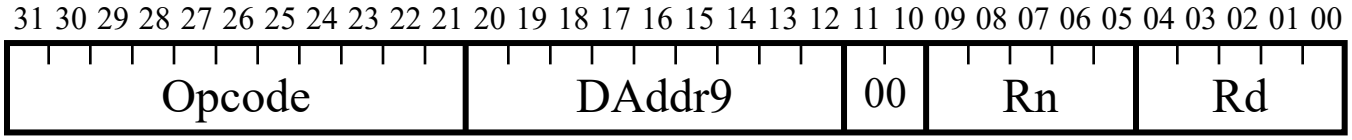
0x244: ADDI
0x248: ANDI
0x164: ADDIS
0x168: ORRI
0x344: SUBI
0x348: EORI
0x2C4: SUBIS
0x2C8: ANDIS

```
ADDI     X8, X3, #35     // X8 = X3+35
```



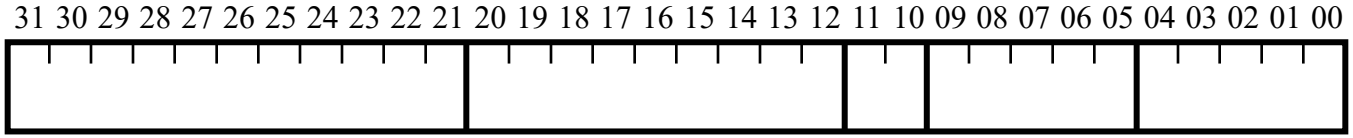
D-Type

Used for memory accesses



0x1C0: STURB
0x1C2: LDURB
0x7C0: STUR
0x7C2: LDUR

```
LDUR     X6, [X15, #12]     // X6 = Memory[X15+12]
```



Conversion example

Compute the sum of the values 0…N-1

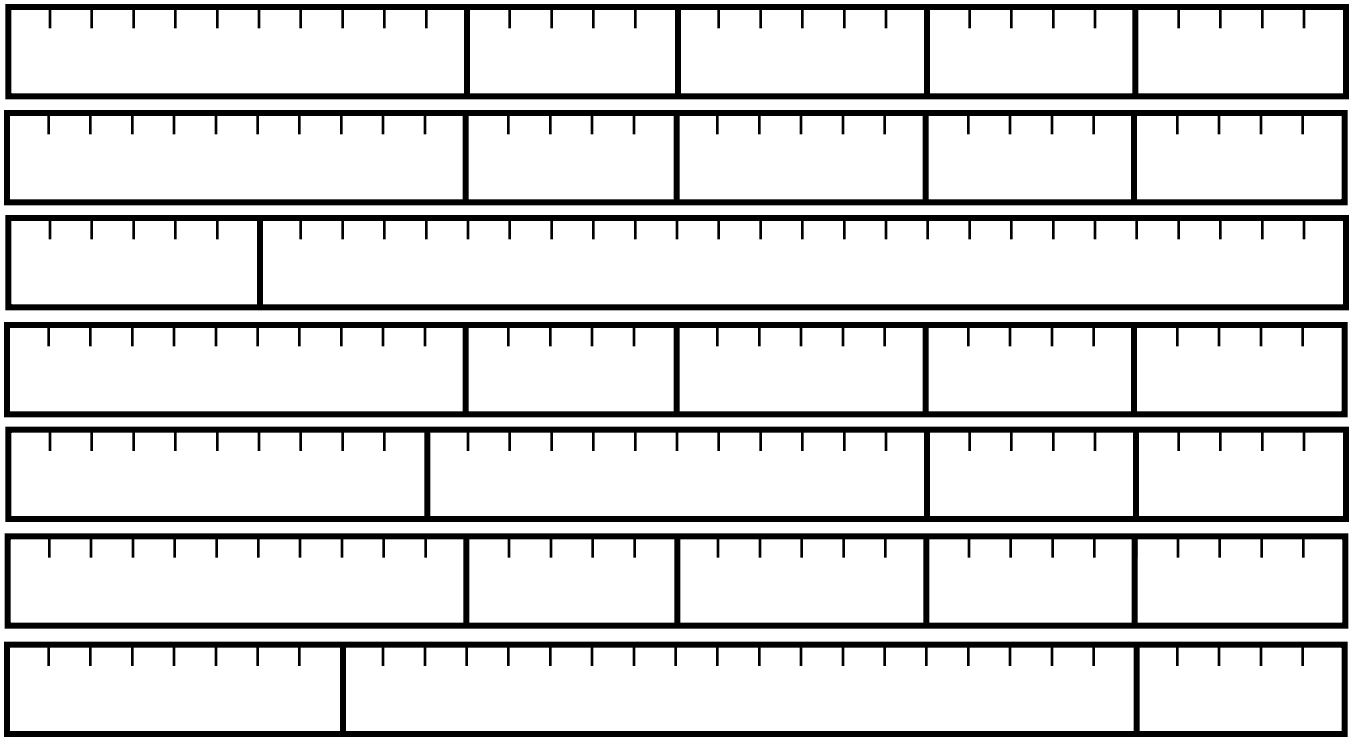
```
   ADD X1, X31, X31

   ADD X2, X31, X31

   B TEST
TOP:
   ADD X1, X1, X2

   ADDI X2, X2, #1
TEST:
   SUBS X31, X2, X0

   B.LT TOP
END:
```



Assembly & Machine Language

Assembly

Machine Language