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

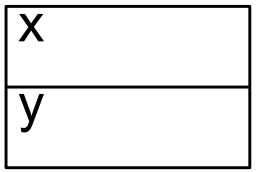
- Understand the basics of assembly language
- Help figure out what the processor needs to be able to do

Not our goal to teach complete assembly/machine language programming

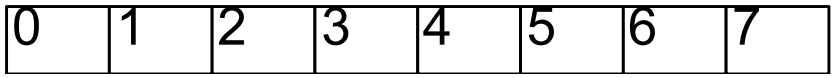
- Floating point
- Procedure calls
- Stacks & local variables

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;  // declare pointer
index = scores; // equivalent to index = &scores[0];
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>
```

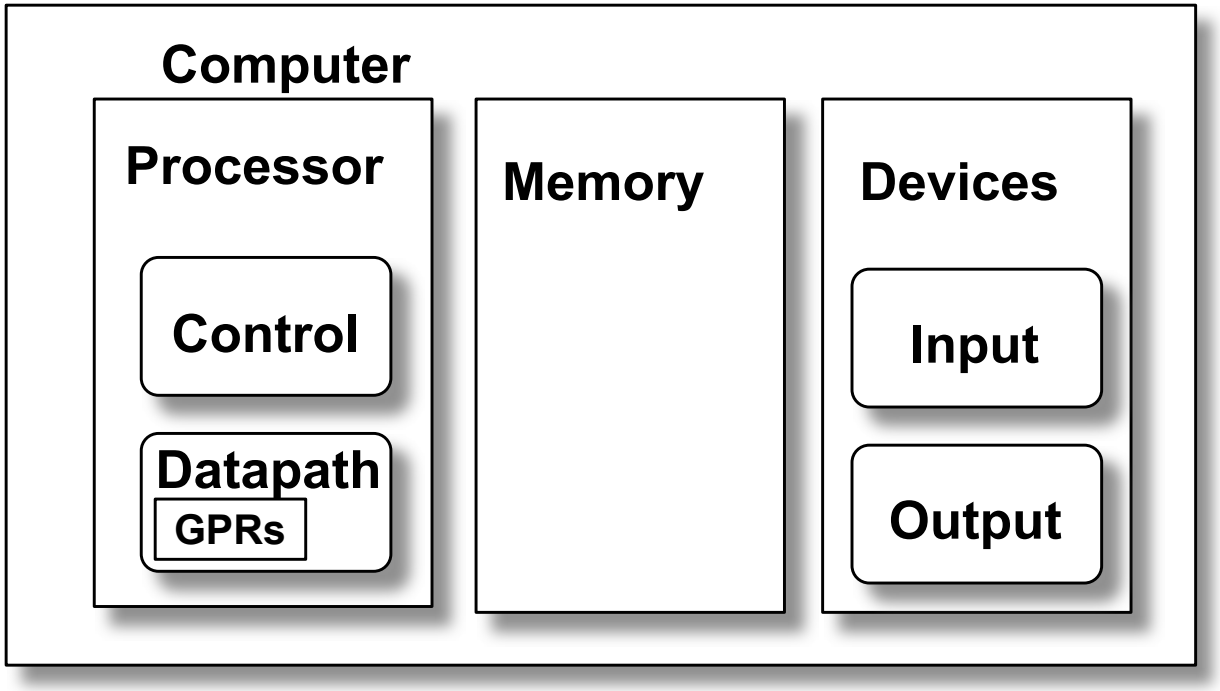
Simple format: easy to implement in hardware

More complex: `A = B + C + D – E`

```
LDUR X2, B
LDUR x3, C
ADD X1, X2, X3  // assumes B is in X2, C is in X3
ADD X1, X1, X4   // assumes D is in X4
SUB X1, X1, X5   // assumes E is in X5 and A is left in X1
STUR X1, A

int foo(int x) { int j = x * 2; return j; }
```

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 (one 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:

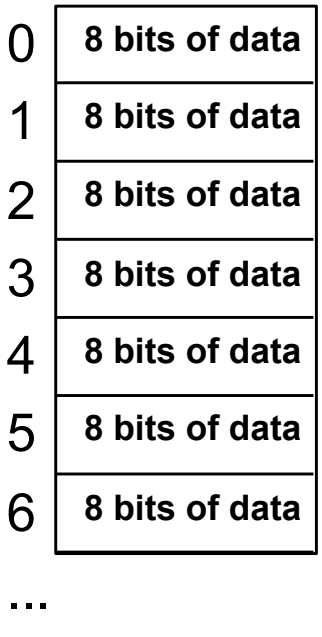
```
x1 = (x0 >> 3) & 0x7    // in C
LSR x1, x0, #3
ANDI x1, x1, #7
```

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.

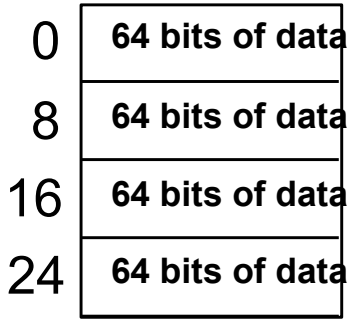


Memory Organization (cont.)

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

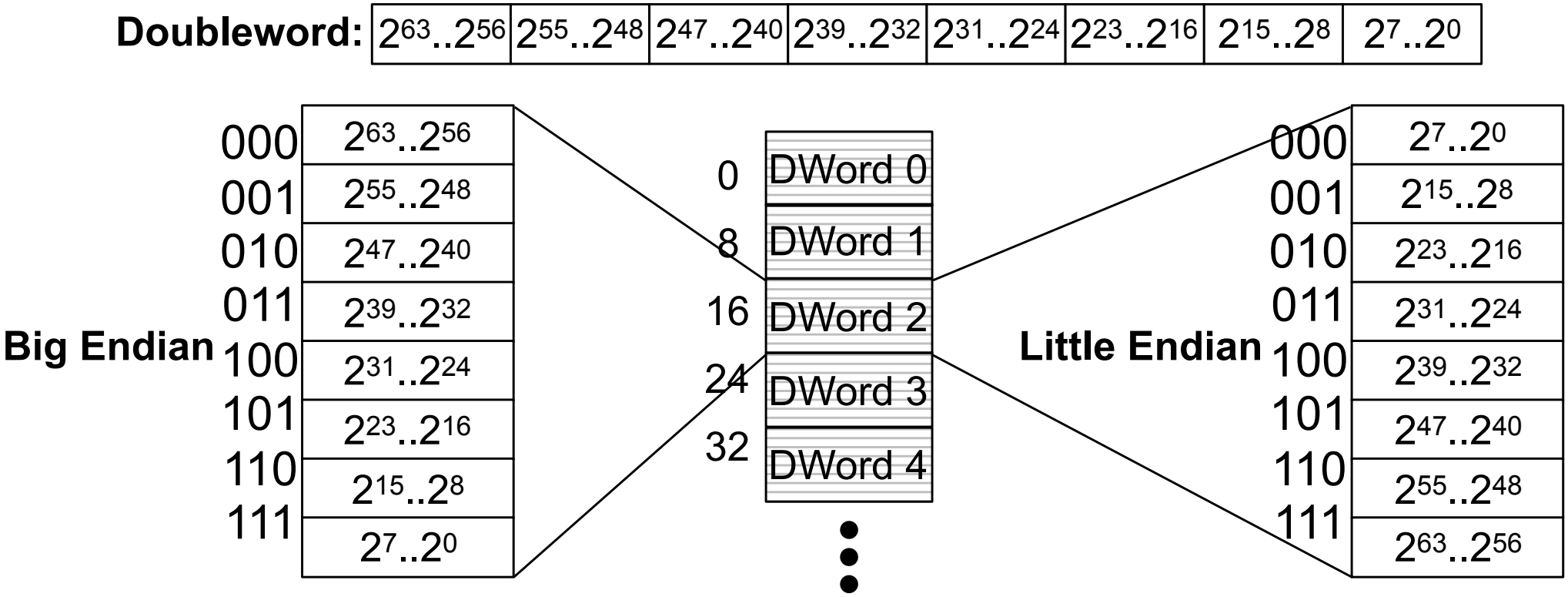
2^{64} bytes with byte addresses from 0 to 2^{64}-1

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

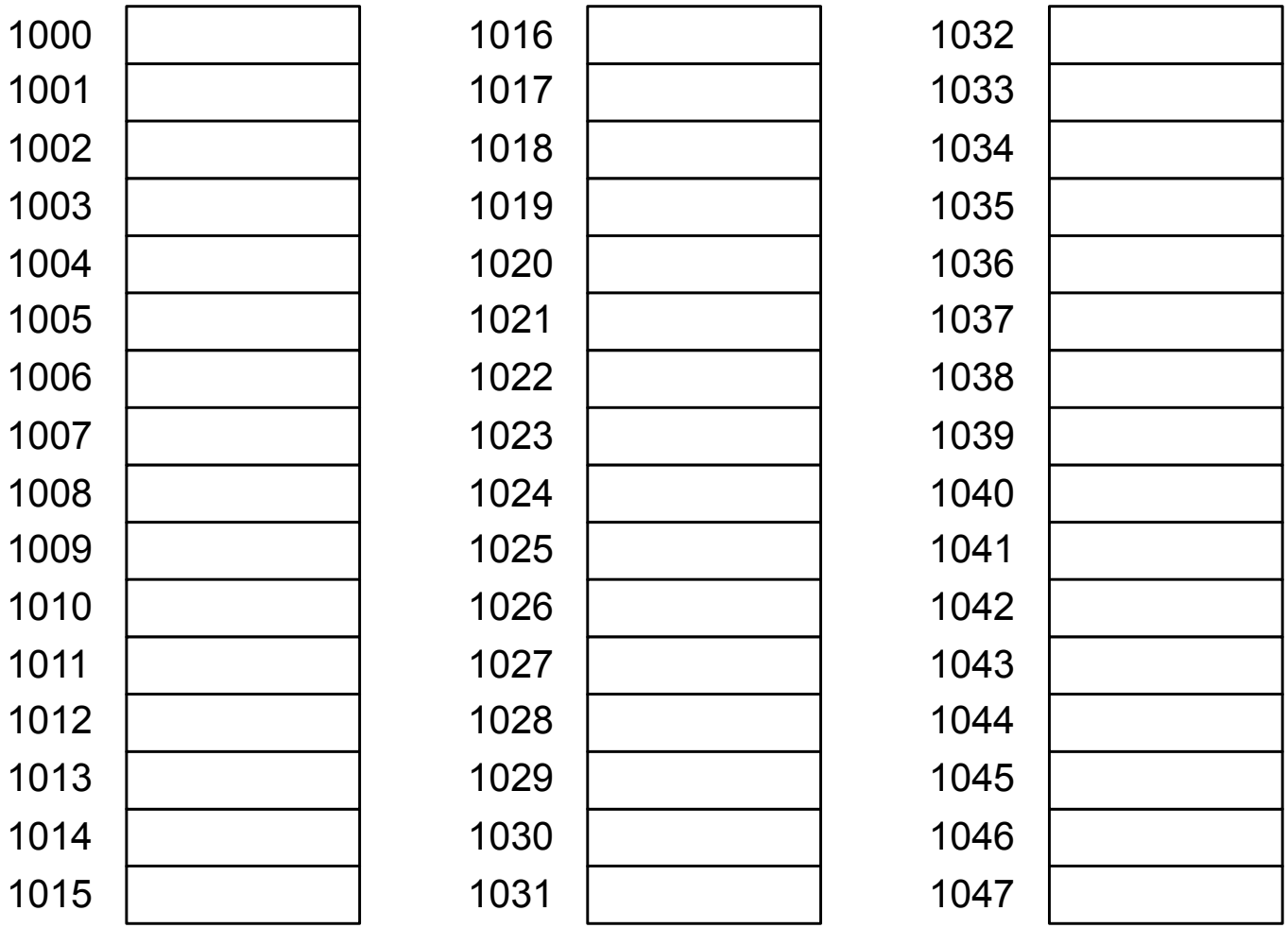
Characters: 8 bits (byte)

Integers: 64 bits (D-word)

Array: Sequence of locations

Pointer: Address (64 bits)

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



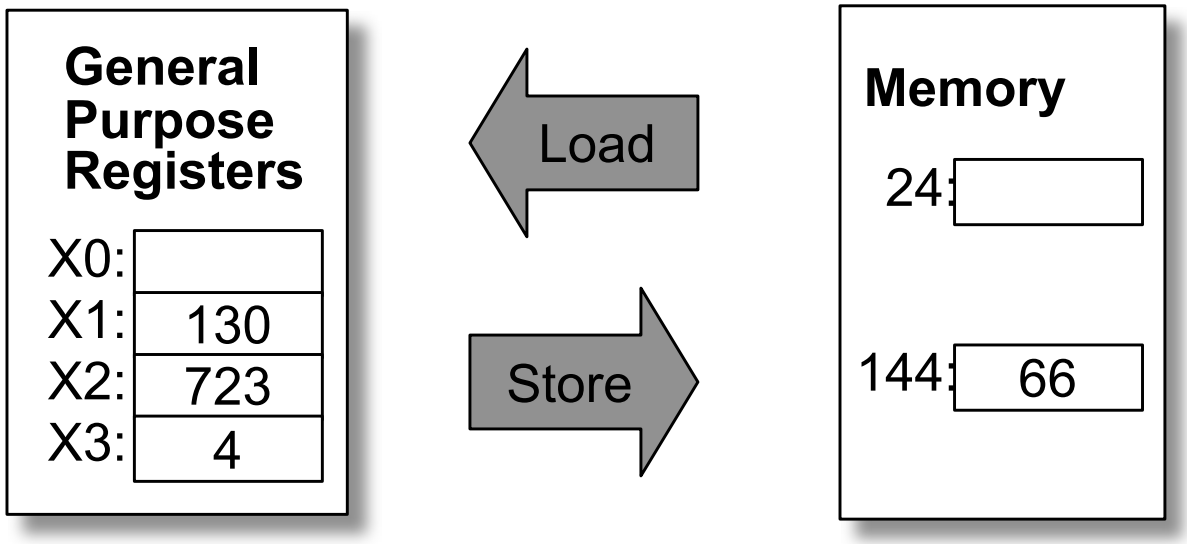
(Note: real compilers place local variables (the "stack") at the top of memory, new'ed structures (the "heap") from near but not at the beginning. We ignore that here for simplicity)

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

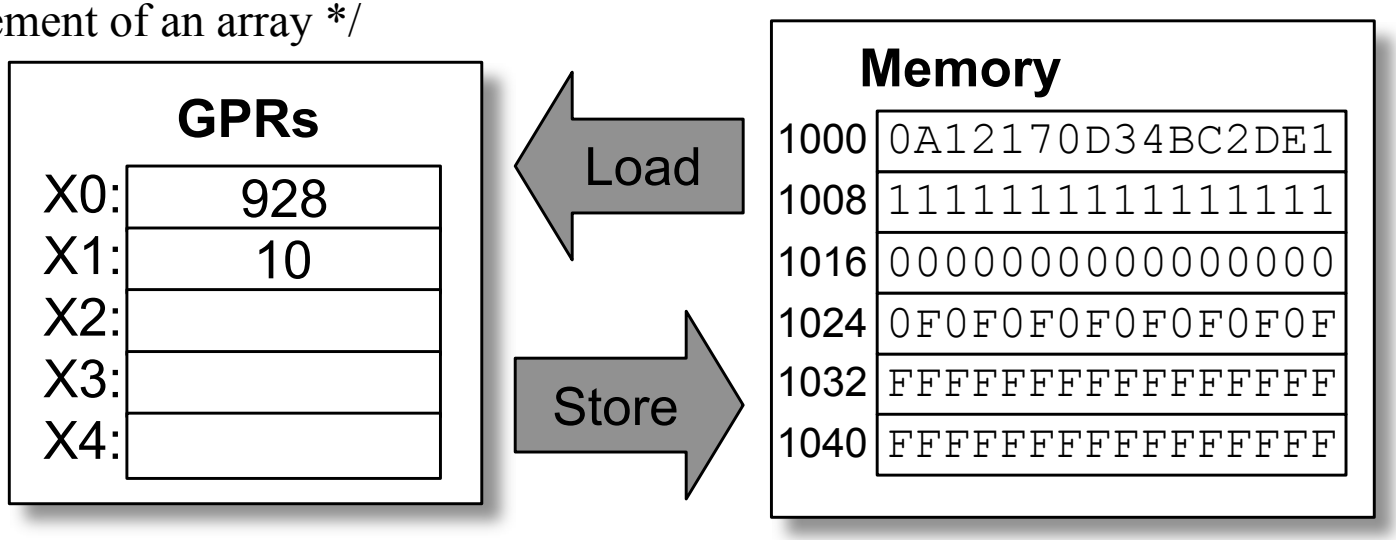
```
LSL x2, x1, #3 ; x2 = x2 * 8
ADD x6, x0, x2;   X6 = &Array[0] + X1
LDUR x7, [x6] ; X7 = Array[x1]
```

NOT ARM! x86: MOV r15, [r14 + r13 * 8]

Stretch!

Array Example

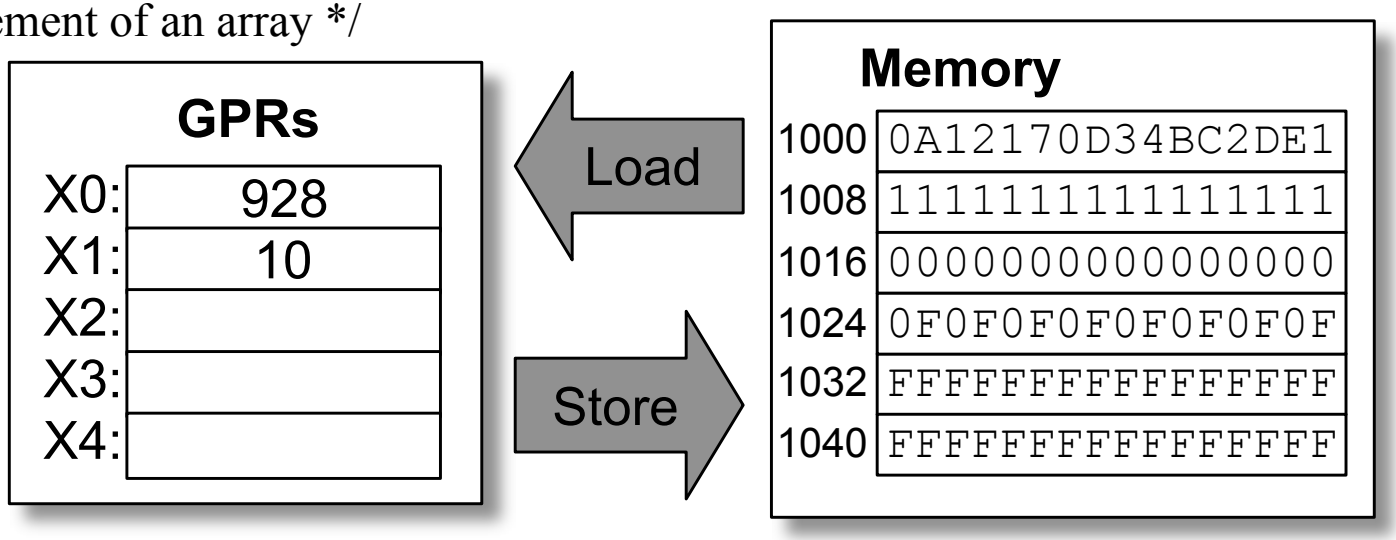
```
/* Swap the kth and (k+1)th element of an array */
swap(int v[], int k) {
      int temp = v[k];
      v[k] = v[k+1];
      v[k+1] = temp;
}

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



Array Example

V[0]=mem[X0]=mem[928] V[1]=mem[X0+8]=mem[936]
V[k]=mem[X0+8*k] V[k+1]=mem[X0+8(k+1)]=mem[X0+8k+8]

```
/* 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
```




```
SWAP:
      LSL    X2, X1, #3      // Compute address of v[k]
      ADD    X2, X0, X2      // get v[k]
      LDUR   X3, [X2, #0]    // get v[k+1]
      LDUR   X4, [X2, #8]    // save new value to v[k]
      STUR   X4, [X2, #0]    // save new value to v[k+1]
      STUR   X3, [X2, #8]
```

```
1101 0011 0110 0000 0000 1100 0010 0010
1000 1011 0000 0010 0000 0000 0000 0010
1111 1000 0100 0000 0000 0000 0100 0011
1111 1000 0100 0000 1000 0000 0100 0100
1111 1000 0000 0000 0000 0000 0100 0100
1111 1000 0000 0000 1000 0000 0100 0011
```

Execution Cycle Example

PC: Program Counter

IR: Instruction Register

Note:
Word addresses
Instructions are 32b

General Purpose Registers

X0:	928
X1:	10
X2:	
X3:	
X4:	

PC:

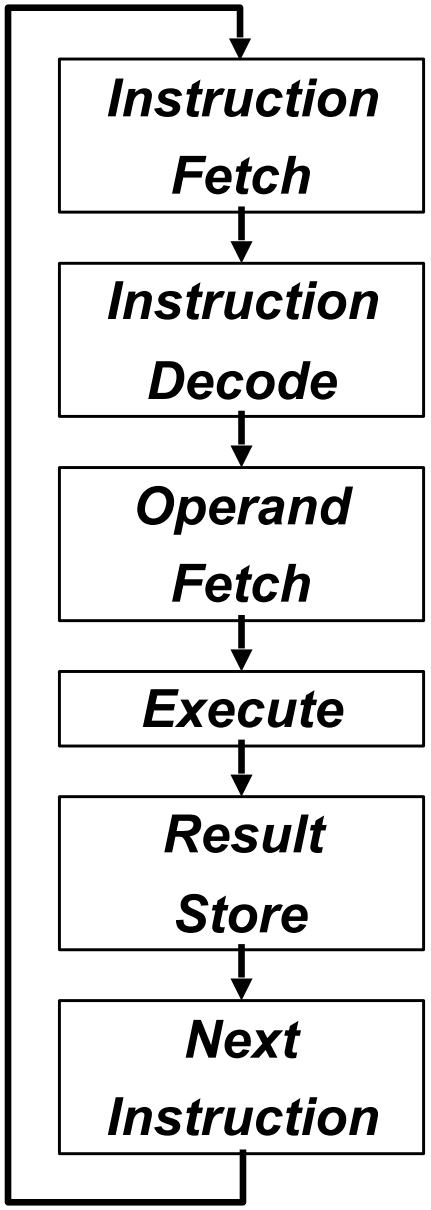
IR:

Load

Store

Memory

0000	D3600C22
0004	8B020002
0008	F8400043
0012	F8408044
0016	F8400044
0020	F8408043
1000	0A12170D34BC2DE1
1008	1111111111111111
1016	0000000000000000
1024	0F0F0F0F0F0F0F0F
1032	FFFFFFFFFFFFFFFF
1040	FFFFFFFFFFFFFFFF



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
CMP 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
```

```
        add   x1, x31, x31    ; sum = 0
        add   x2, x31, x31    ; i = 0
loop:                         ; for(int I = 0; I != N; I++)
        cmp   x2, x0          ; i != N ?
        b.eq  end_loop
        add   x1, x2, x1      ; sum += i
        addi  x2, x2, #1      ; i++
        b     loop
end_loop:
```

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
   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:
```

```
// X0 = N, X1 = sum, X2 = I
   ADD X1, X31, X31 // sum = 0
   ADD X2, X31, X31 // I = 0
   B TEST           // Test@bottom
TOP:
   ADD X1, X1, X2   // sum += I
   ADDI X2, X2, #1  // I++
TEST:
   CMP X2, X0       // Check I vs N
   B.LT TOP         // if (I<N) cont.
END:
```

Note: Can you do the loop with less # of branches per iteration?

Branch at bottom of loop, branching back. Branch forward at top to this branc

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
 the_while:
    ldurb   x1, [x0]        ; x1 = *index
    cbz     x1, end_while   ; while (*index != 0)
    cmp     x1, #97         ; if (*index < 'a'…)
    b.lt    is_upper
    cmp     x1, #122        ; if (*index > 'z'….)
    b.gt    is_upper
    sub     x1, x1, #32     ; x1 = x1 - 'a' + 'A'
    sdurb   x1, [x0]        ; *index = x1
 is_upper:
    addi    x0, x0, #1      ; index++
    b       the_while
 end_while:
```

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
    LDUR   X0, [X31, #80]    // index = string
LOOP:
    LDURB  X1, [X0, #0]      // load byte *index
    CBZ    X1, END           // exit if *index == 0
    CMPI   X1, #97           // is *index < 'a'?
    B.LT   NEXT              // don't change if < 'a'
    CMPI   X1, #122          // is *index > 'z'?
    B.GT   NEXT              // don't change if > 'z'
    SUBI   X1, X1, #32       // X1 = *index + ('A' - 'a')
    STURB  X1, [X0, #0]      // *index = new value;
NEXT:
    ADDI   X0, X0, #1        // index++;
    B      LOOP              // continue the loop
END:
```

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
    ADD     X9, X0, X9      // Compute address of v[k]
    LDUR    X10, [X9, #0]   // get v[k]
    LDUR    X11, [X9, #8]   // get v[k+1]
    STUR    X11, [X9, #0]   // save new value to v[k]
    STUR    X10, [X9, #8]   // save new value to v[k+1]
    BR      X30             // return from subroutine
```

```
11010011011 00000 000011 00001 01001
10001011000 01001 000000 00000 01001
11111000010 000000000 00 01001 01010
11111000010 000001000 00 01001 01011
11111000000 000000000 00 01001 01011
11111000000 000001000 00 01001 01010
11010110000 00000 000000 00000 11110
```

Stretch!

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:
```

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      END = +9
    CMPI   X1, #97
    B.LT   NEXT         NEXT = +5
    CMPI   X1, #122
    B.GT   NEXT         NEXT = +3
    SUBI   X1, X1, #32
    STURB  X1, [X0, #0]
NEXT:
    ADDI   X0, X0, #1
    B      LOOP
END:                    LOOP = -9
```

Instruction Types

Can group instructions by # of operands

3-register

2-register

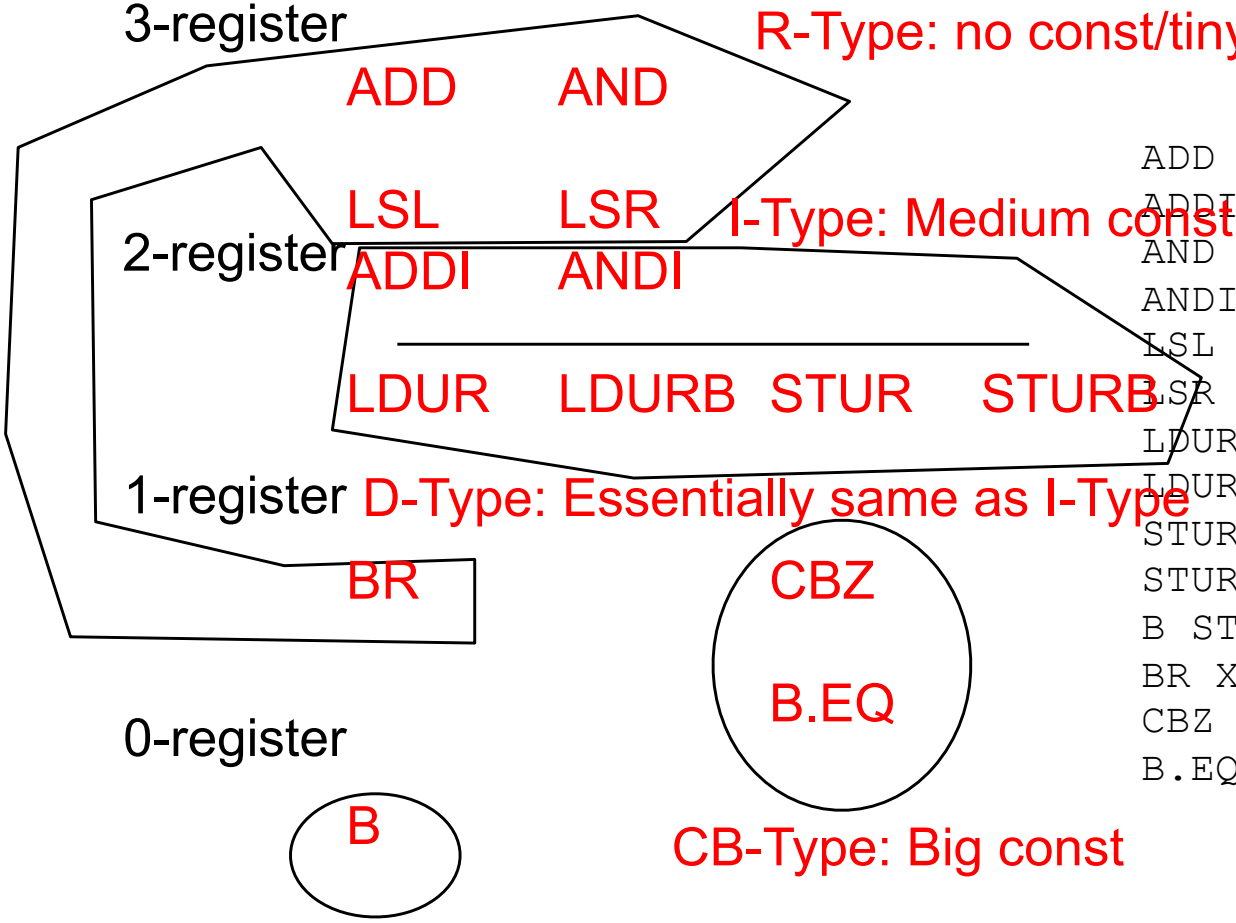
1-register

0-register

```
ADD X0, X1, X2
ADDI X0, X1, #100
AND X0, X1, X2
ANDI X0, X1, #7
LSL X0, X1, #4
LSR X0, X1, #2
LDUR X0, [X1, #14]
LDURB X0, [X1, #14]
STUR X0, [X1, #14]
STURB X0, [X1, #14]
B START
BR X30
CBZ X0, FOO
B.EQ DEST
```

Instruction Types

Can group instructions by # of operands



R-Type: no const/tiny const

I-Type: Medium const

D-Type: Essentially same as I-Type

CB-Type: Big const

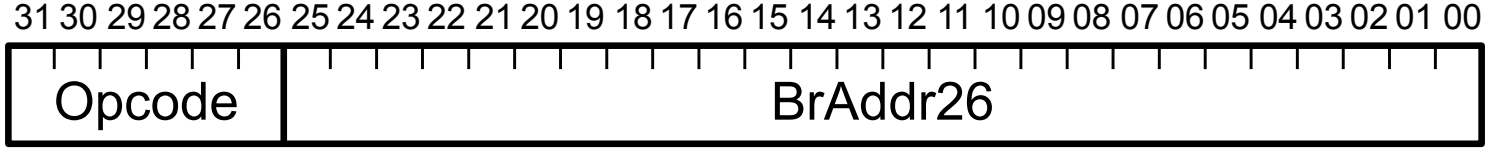
B-Type: Almost all const

```
ADD X0, X1, X2
ADDI X0, X1, #100
AND X0, X1, X2
ANDI X0, X1, #7
LSL X0, X1, #4
LSR X0, X1, #2
LDUR X0, [X1, #14]
LDURB X0, [X1, #14]
STUR X0, [X1, #14]
STURB X0, [X1, #14]
B START
BR X30
CBZ X0, FOO
B.EQ DEST
```

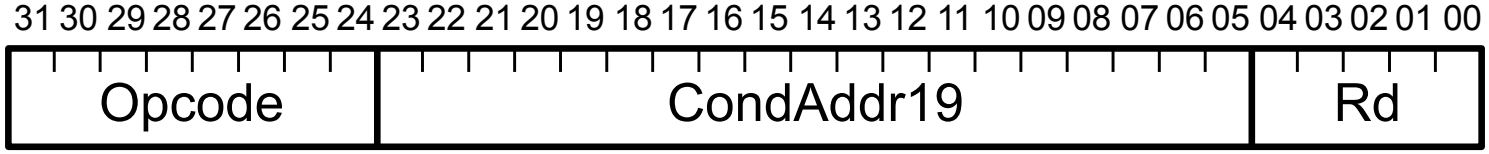
Instruction Formats

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

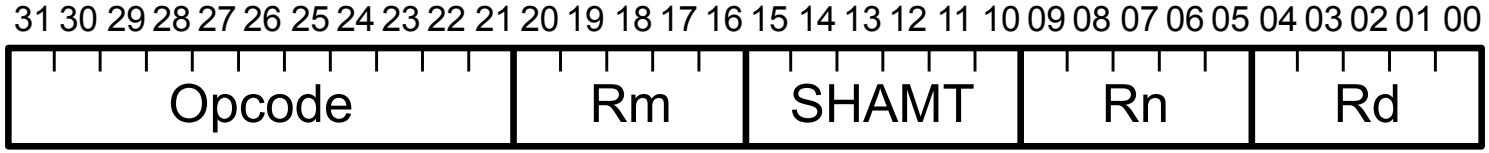
Branch (B-Type) Instr[31:21] = 0A0-0BF



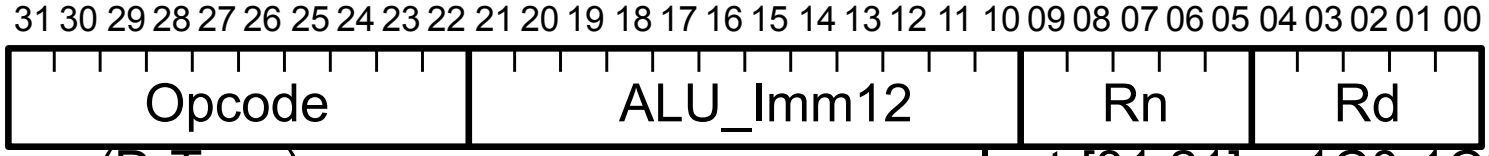
Conditional Branch (CB-Type) Instr[31:21] = 2A0-2A7, 5A0-5AF



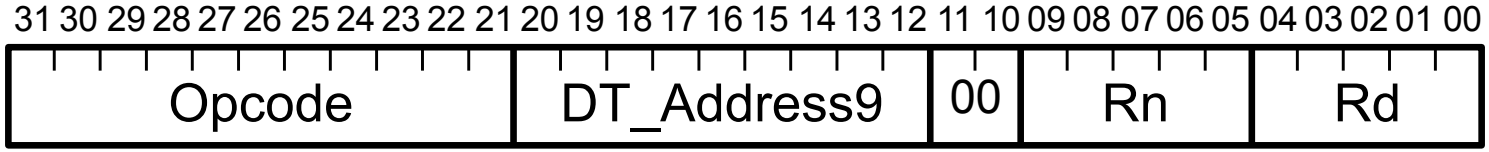
Register (R-Type) Instr[31:21] = 450-458, 4D6-558, 650-658, 69A-758



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

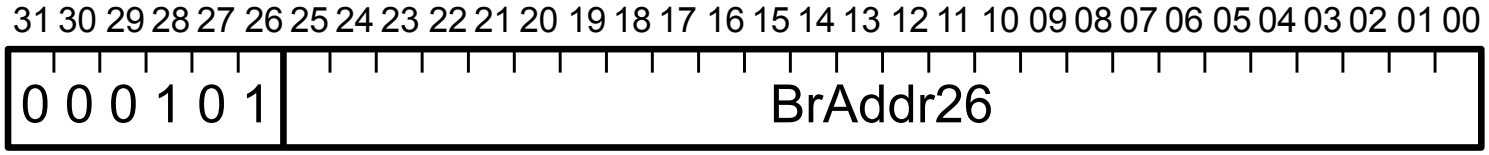


Memory (D-Type) Instr[31:21] = 1C0-1C2, 7C0-7C2



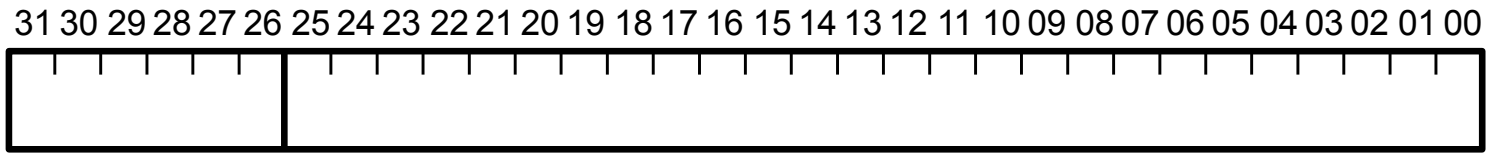
B-Type

Used for unconditional branches



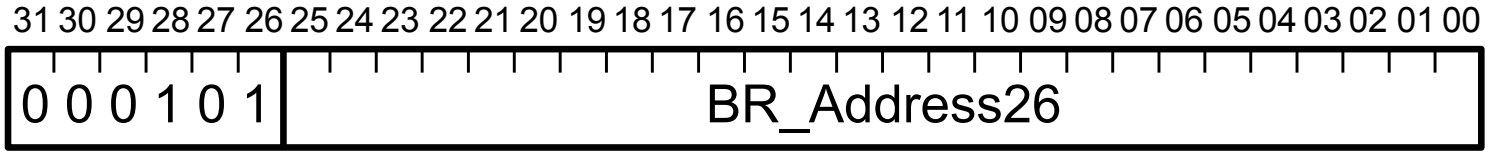
0x05: B

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



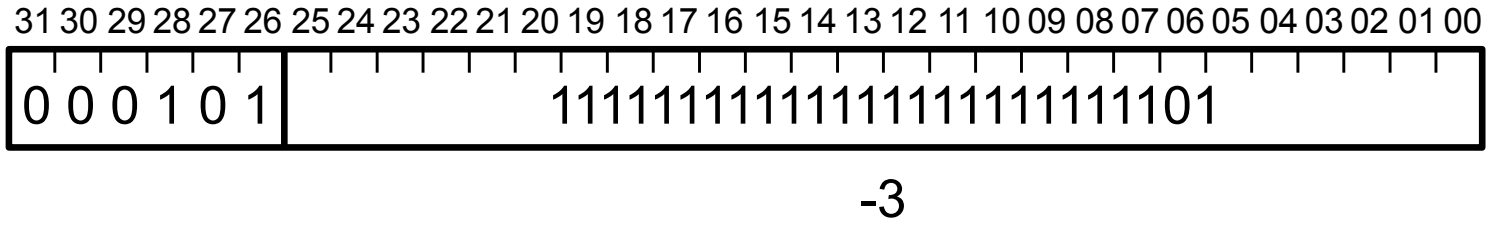
B-Type

Used for unconditional branches



05: B

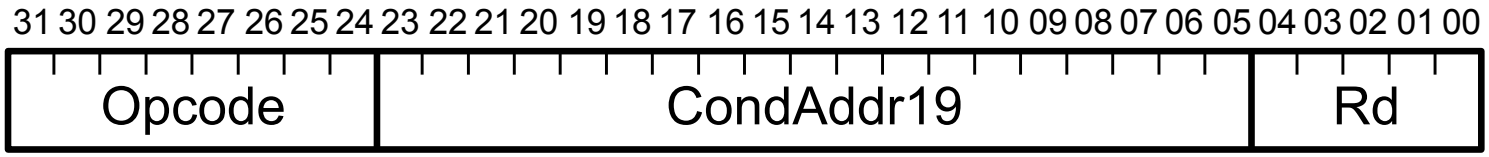
```
B -3      // PC = PC + 4*-3
```



-3

CB-Type

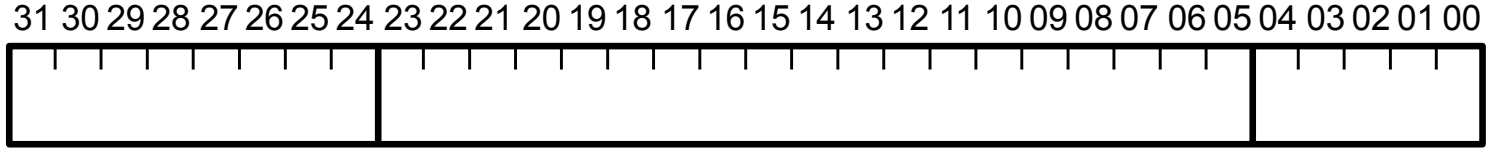
Used for conditional branches



Reg or Cond. Code

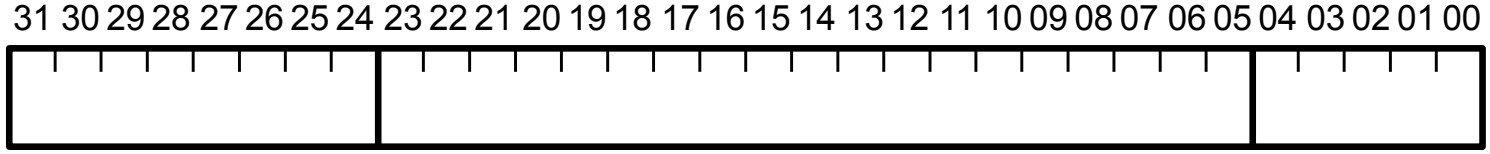
0x54: B.cond
0xB4: CBZ
0xB5: CBNZ

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



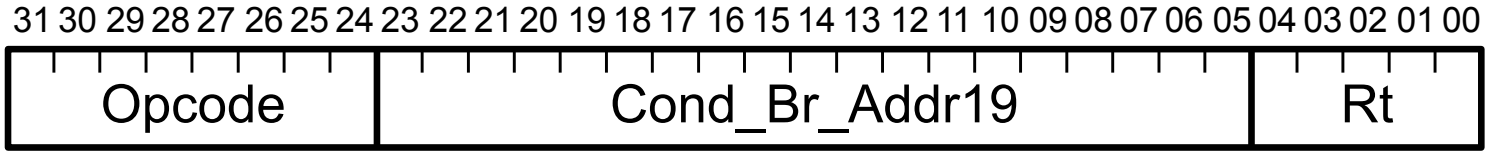
Condition Codes
0x00: EQ (==)
0x01: NE (!=)
0x0A: GE (>=)
0x0B: LT (<)
0x0C: GT (>)
0x0D: LE (<=)

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



CB-Type

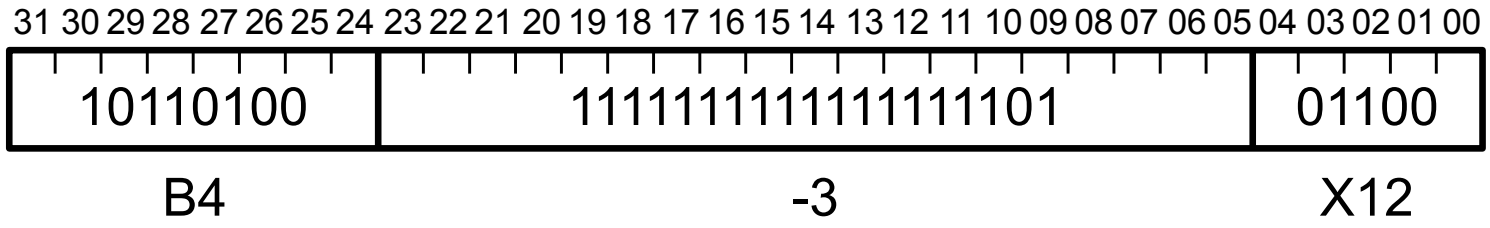
Used for conditional branches



Reg or Cond. Code

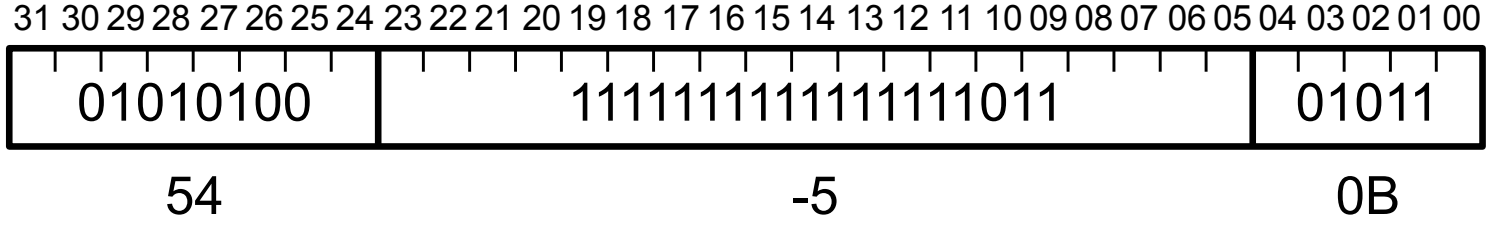
54: B.cond
B4: CBZ
B5: CBNZ

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



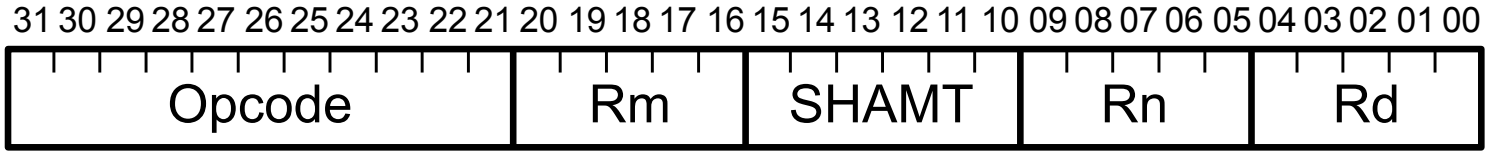
Condition Codes
00: EQ (==)
01: NE (!=)
0A: GE (>=)
0B: LT (<)
0C: GT (>)
0D: LE (<=)

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



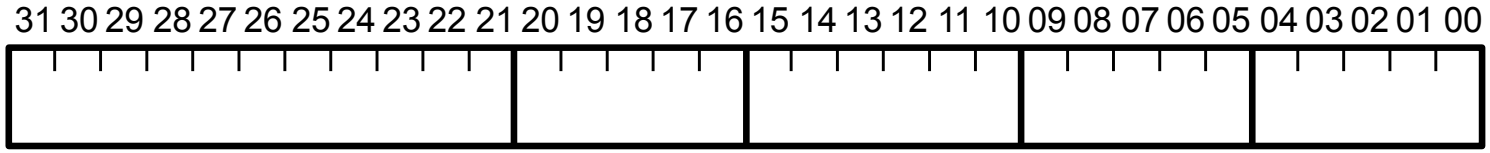
R-Type

Used for 3 register ALU operations and shift

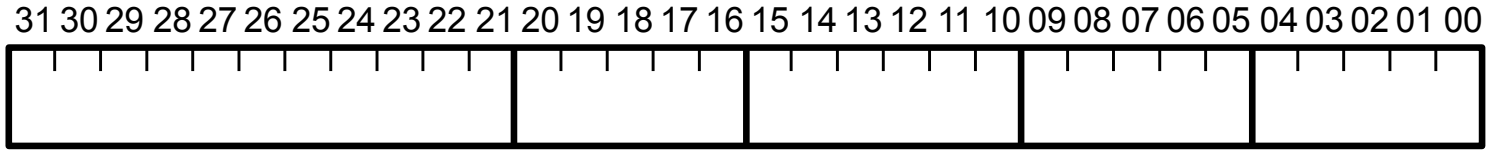


0x450: AND
0x458: ADD
0x4D6: SDIV, shamt=02
0x4D8: MUL, shamt=1F
0x550: ORR
0x558: ADDS
0x650: EOR
0x658: SUB
0x69A: LSR
0x69B: LSL
0x6B0: BR, rest all 0's but Rd
0x750: ANDS
0x758: SUBS

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

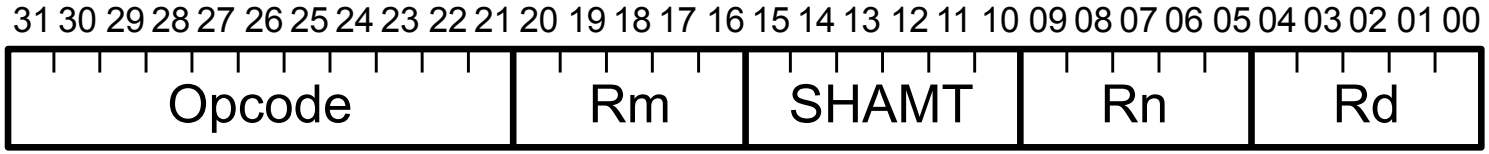



```
LSL      X10, X4, #6       // X10 = X4<<6
```



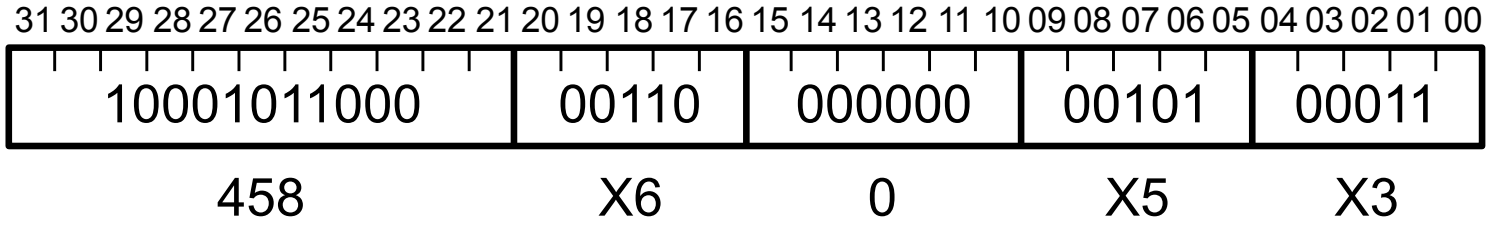
R-Type

Used for 3 register ALU operations and shift

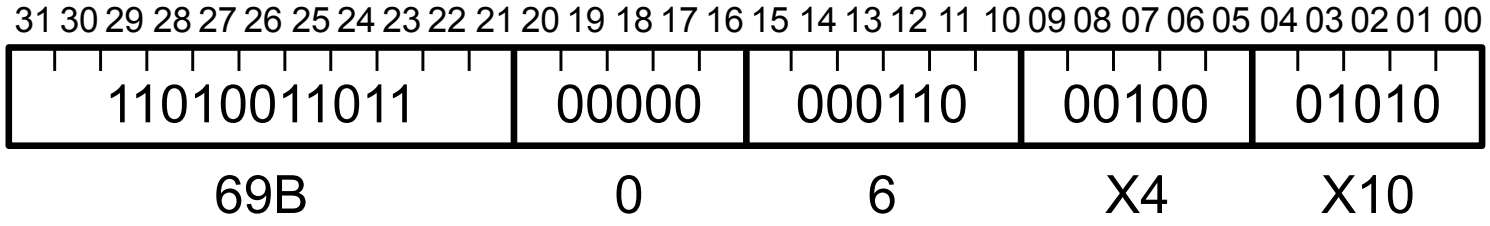


450: AND
458: ADD
4D6: SDIV, shamt=02
4D8: MUL, shamt=1F
550: ORR
558: ADDS
650: EOR
658: SUB
69A: LSR
69B: LSL
6B0: BR, rest all 0's but Rd
750: ANDS
758: SUBS

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

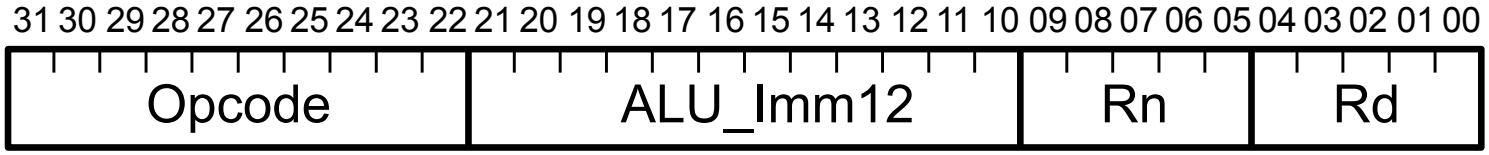



```
LSL      X10, X4, #6       // X10 = X4<<6
```



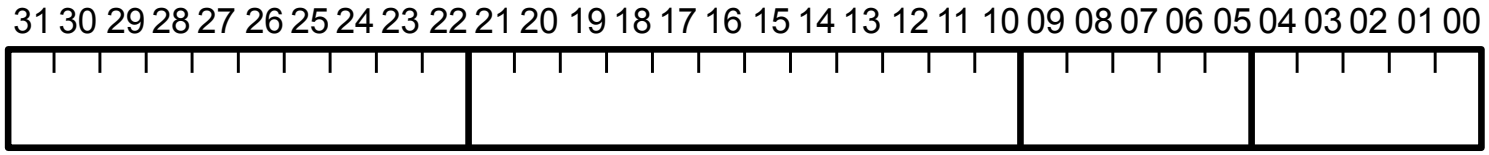
I-Type

Used for 2 register & 1 constant ALU operations



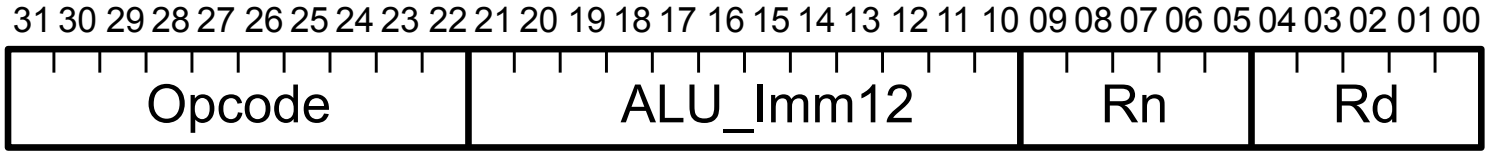
0x244: ADDI
0x248: ANDI
0x164: ADDIS
0x168: ORRI
0x344: SUBI
0x348: EORI
0x2C4: SUBIS
0x2C8: ANDIS

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



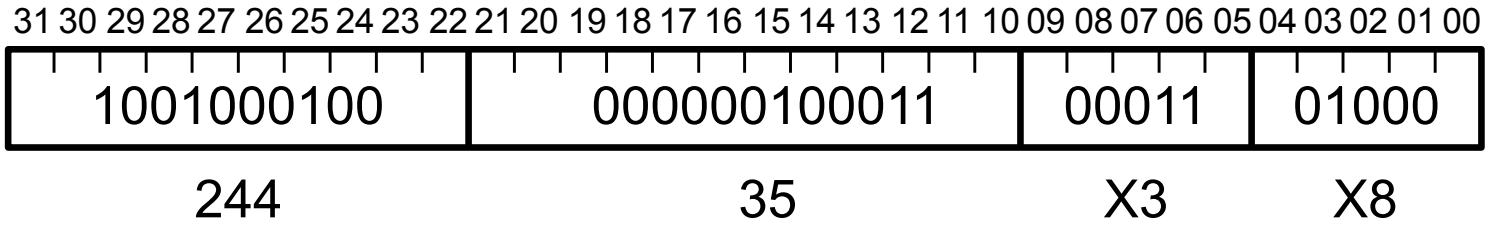
I-Type

Used for 2 register & 1 constant ALU operations



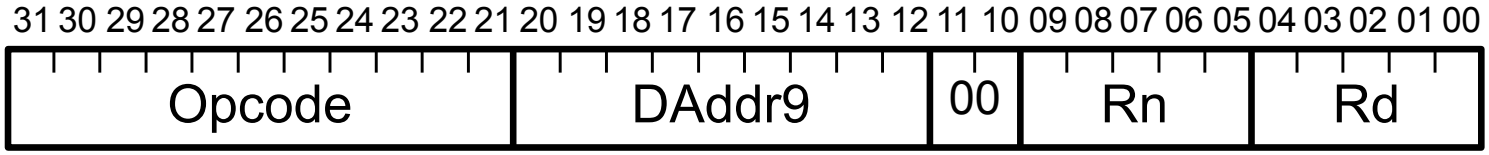
244: ADDI
248: ANDI
164: ADDIS
168: ORRI
344: SUBI
348: EORI
2C4: SUBIS
2C8: ANDIS

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



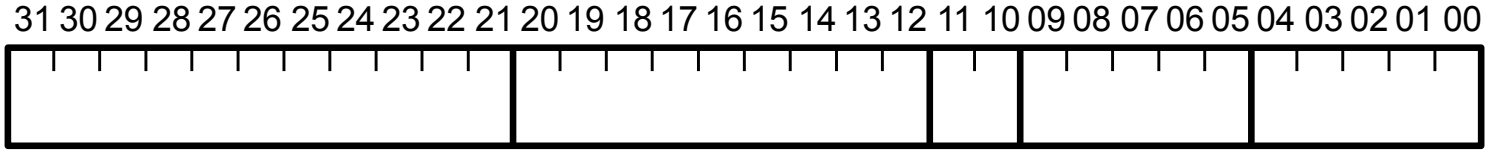
D-Type

Used for memory accesses



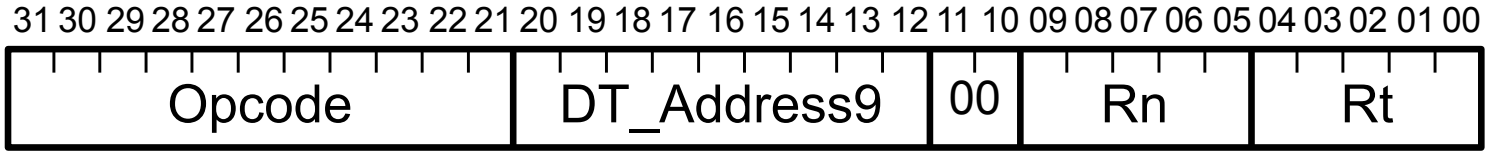
0x1C0: STURB
0x1C2: LDURB
0x7C0: STUR
0x7C2: LDUR

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



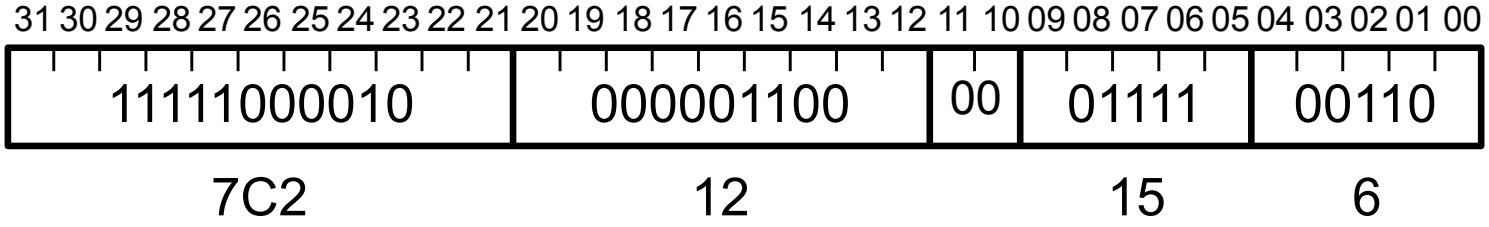
D-Type

Used for memory accesses



1C0: STURB
1C2: LDURB
7C0: STUR
7C2: LDUR

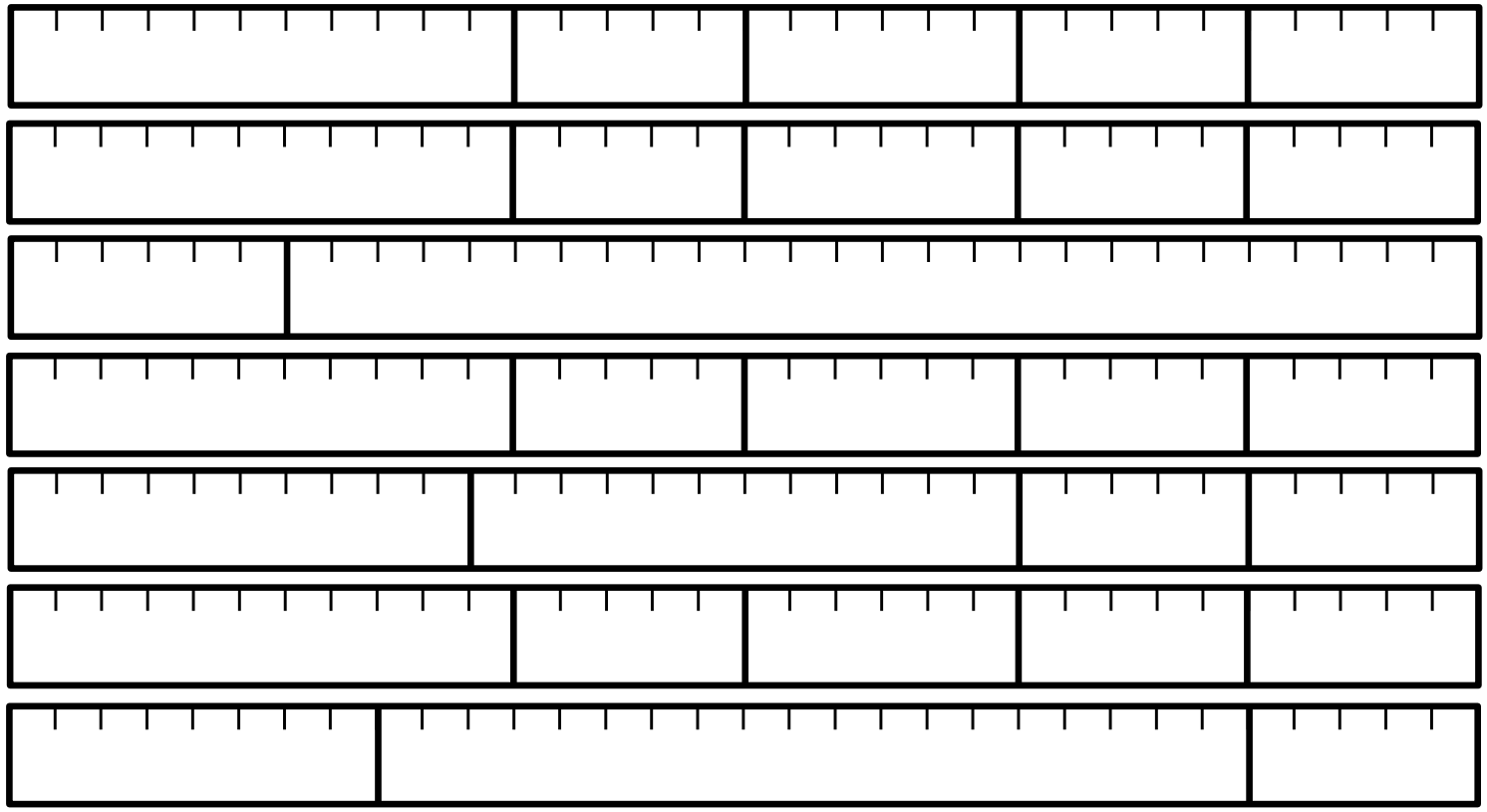
```
LDUR     X6, [X15, #12]        // X6 = Memory[X15+12]
```



Conversion example

Compute the sum of the values 0…N-1

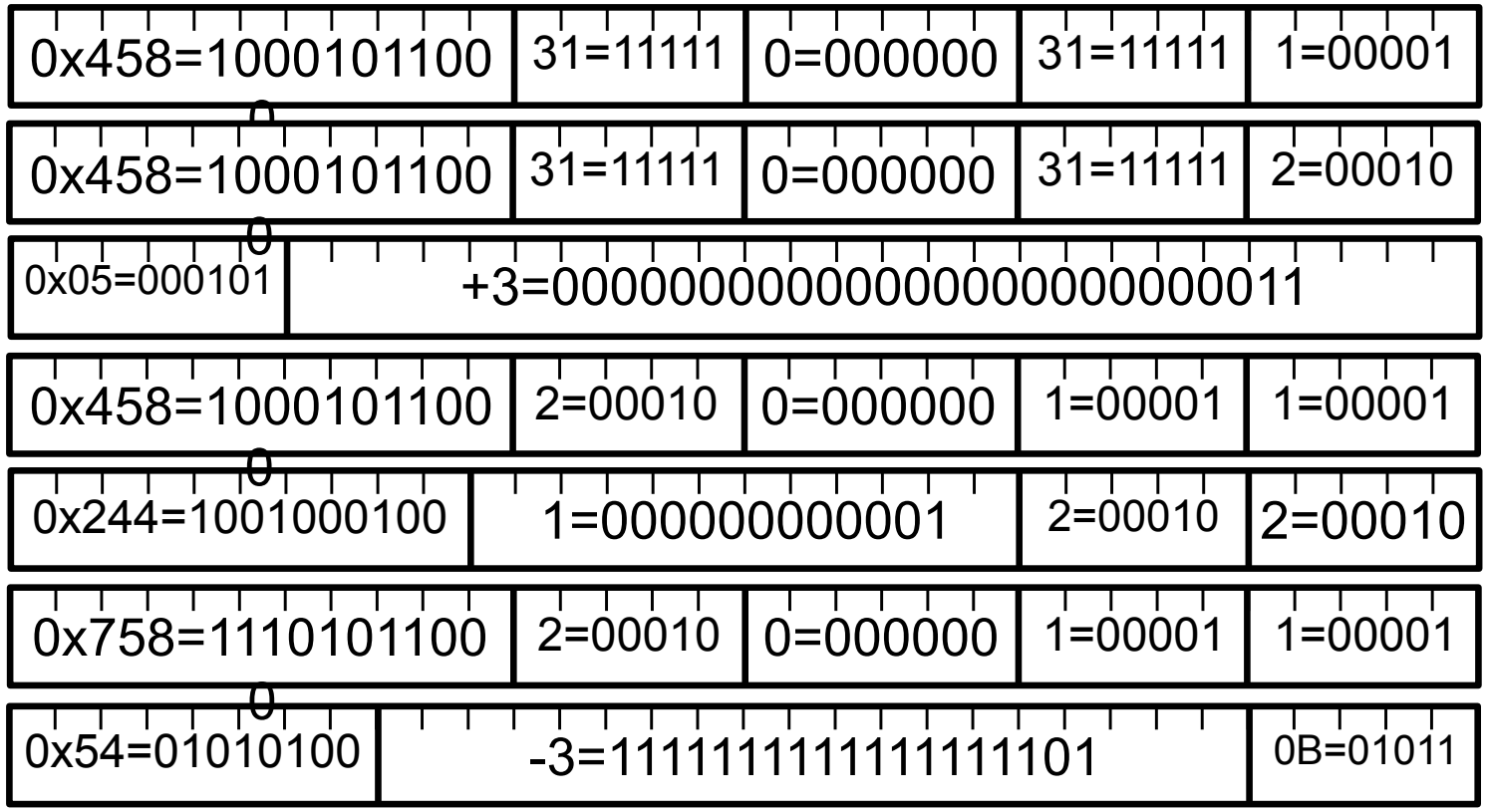
```
   ADD X1, X31, X31

   ADD X2, X31, X31

   B TEST
TOP:
   ADD X1, X1, X2

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

   B.LT TOP
END:
```



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

- Simple instructions
- Mnemonics for human developers
- (Almost) 1-to-1 relationship to machine language

Machine Language

- Numeric representation of instructions
- Fixed format(s), simple encode & decode
- Directly control microprocessor hardware