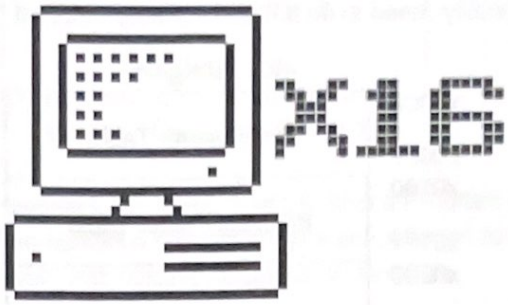
# X16




## Introduction

The X16[1] is a 16-bit computer similar to early computers like the DEC PDP-11 or the Intel 8086 that was used in the first IBM PC.

In this assignment, we will implement an **emulation** of the X16. Your emulator will run X16 programs, enough to run simple programs, and even play a maze navigation game. In addition, you will write an **assembler** for the X16, so that you can compile assembler code written in X16 assembler into binary objects that can load and be run by the emulator.

## Architecture

A 16 bit integer can store $2^{16}$ = 65536 different values. The X16's memory consists of 65536 16-bit words. Each instruction is also 16 bits, and is specified in the X16 Instruction Set Architecture reference. The instruction set is simple compared to the x86 or even the 8086, but it has all the important instructions that modern CPUs have.

### Memory Layout

The diagram below shows the memory layout of X16. There are 65536 different words in memory. Each memory location is word addressable - that is, you can only address a 16-bit word, not a byte within a word.

---

[1] X16 is a simplified computer that elides supervisor, interrupt and stack modes. Credit goes to Yale Patt at UTA and Sanjay Patel at UIUC for the original concept.

The operating system runs in location 0x0000 to 0xffff. Location 0xfe00 to 0xffff is used to memory map IO devices. That is, memory locations in that range correspond to IO devices. Reading one memory region in that range will return a value indicating whether a key has been pressed, while reading another memory region in that range will return the actual character that was typed. The X16 emulator we will write will not have any OS or interrupt vector table. The trap vector table functionality (used to do IO) and memory mapped IO is implemented for you.

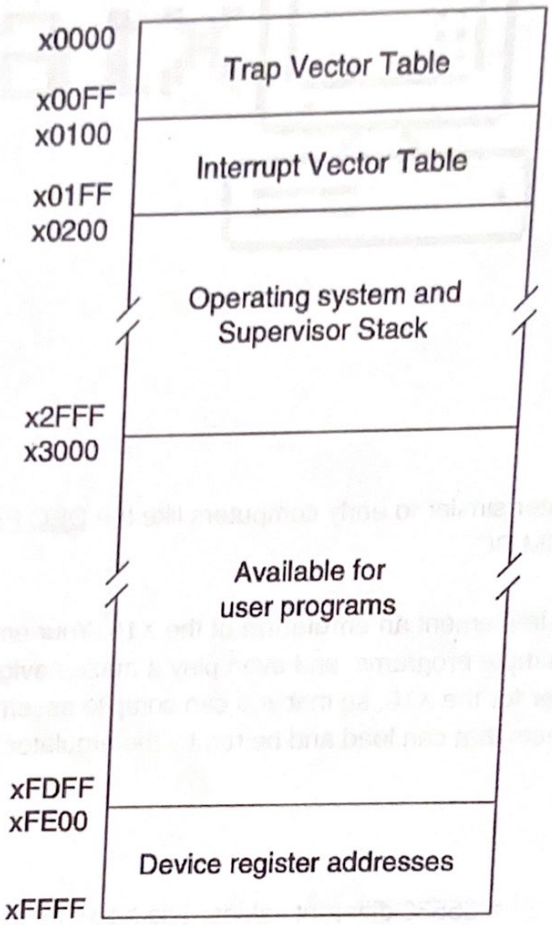



Figure 1. X16 memory layout.

To simplify things, we will focus our attention on **user programs**. User programs are loaded at 0x3000. The PC is set to 0x3000 to start executing the user program. The program executes until it sees a HALT instruction, at which point the program terminates and your emulation comes to an end.

## Register File

The X16 has a register file that controls an array of CPU registers.

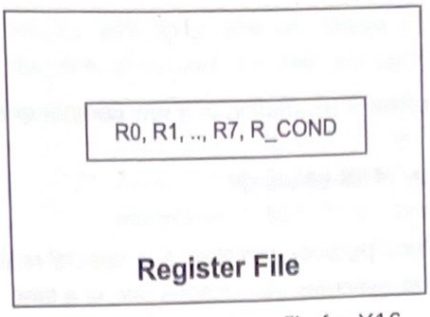



Figure 2. The register file for X16

There are 8 general purpose registers, R0 to R7. There is also a Program Counter (PC) and a Condition Register (COND). The condition register holds a set of codes to indicate whether the result of the last register operation was positive, negative or zero.

## CPU Instructions

The full instruction set is specified in the X16 Instruction Set Architecture reference.

### Example: ADD immediate

Here is an example of the ADD instruction.

```
ADD R3, R2, 10
```

This adds the contents of R2 to the value 10, and puts the result in R3. In X16, and unlike Intel AT&T syntax, the destination is the first argument to most instructions.

### Example: ST - Store

Here is an example of the ST instruction.

```
ST R5, 42
```

This instruction stores the contents of R5 into the memory location whose address is computed by adding 42 to the PC. For example, if R5 holds the value 19, the PC holds the value 5000, then the value 19 will be stored at memory location 5000 + 42 = 5042.

### Example: JMP - Jump

Here is an example of the JMP instruction.

```
JMP R2
```

This instruction unconditionally jumps to the location specified by the contents of the base register R2. For example, if R2 holds the value 300, then the PC is set to 300.

# Emulator

We will realize the X16 as a software simulation of a few components:

1. The memory as an array of 16-bit words
2. The CPU, as
    a. An array of general purpose registers and special registers
    b. A control unit that executes instructions one at a time

For this assignment, we will emulate a **single cycle X16 in software**. That is, on one clock tick, a single instruction moves from IF to ID to EX to MEM to WB. **Instructions are not pipelined**. On each clock tick, one instruction moves completely through the entire 5-stage sequence.

## Register File

The **Register File** structure holds all the ten registers in the X16 CPU. There are **8 General Purpose registers**, from R0 to R7, the **Program Counter** (R_PC), and the **Condition Register** (R_COND).

```
// 10 total registers, each of which is 16 bits. Most of them are general
// purpose, but a few have designated roles.
typedef enum {
    R_R0 = 0,
    R_R1,
    R_R2,
    R_R3,
    R_R4,
    R_R5,
    R_R6,
    R_R7,
    R_PC,                   // program counter
    R_COND,                 // condition flag
} reg_t;

// There are 10 total registers
#define MAX_REGISTERS   10
```

`R_COND` holds one of three values: `FL_POS`, `FL_ZRO` or `FL_NEG`, depending on whether the last register instruction generated a result that was positive, zero or negative respectively. These condition codes are defined below.

```
// The R_COND register stores condition flags which provide information
// about the most recently executed calculation.
```

```
// This is encoded in 3 bits, and only one of these bits must be set at the
// same time. These 3 bits are also set in the BR instruction.
typedef enum {
    FL_POS = 1,             // Positive - bit 0 is set
    FL_ZRO = 2,             // Zero - bit 1 is set
    FL_NEG = 4,             // Negative - bit 2 is set
} condition_t;
```

## X16 Machine

The X16 machine itself combines memory and registers into one structure.

```
// Total amount of memory for 16 bit address
#define MAX_MEMORY                  65536

// The X16 machine
typedef struct x16 {
    // The memory of the computer is emulated by this array, each slot of
    // which stores a 16 bit value.
    uint16_t memory[MAX_MEMORY];

    // The register file contains R0-R7, PC and condition registers
    uint16_t registers[MAX_REGISTERS];
} x16_t;
```

The entire contents of the memory are in the `memory` field. This contains 65536 16-bit words.

The `registers` field holds the Register File. This array holds all the ten registers in the X16 CPU.

The **x16.h** header defines some convenience functions you can use.

```
// Initialize and return a new x16 machine. The program counter
// is set to the default start location DEFAULT_CODESTART
// All registers and memory are cleared to 0
x16_t* x16_create();

// Free all resources consumed by a machine
void x16_free(x16_t* machine);

// Get the program counter
uint16_t x16_pc(x16_t* machine);
```

```
// Get the condition register
uint16_t x16_cond(x16_t* machine);

// Get the contents of the machine register
uint16_t x16_reg(x16_t* machine, reg_t reg);

// Set the machine register
void x16_set(x16_t* machine, reg_t reg, uint16_t value);

// Read memory. Handles memory mapped IO
uint16_t x16_memread(x16_t* machine, uint16_t address);

// Memory write. Handles memory mapped IO
void x16_memwrite(x16_t* machine, uint16_t address, uint16_t val);

// Get a pointer to the 16 bit word in the given offset in memory
uint16_t* x16_memory(x16_t* machine, uint16_t offset);

// Dump X16
void x16_print(x16_t* machine);
```

The x16_t structure itself is defined in the x16.c file, and thus not visible to other modules. When you manipulate X16, use these functions rather than manipulating the struct directly. There is a useful function that prints out the state of the machine. It will dump the contents of the PC, all the registers, and calculate a hash of the contents of memory.

## Control Unit

The Control Unit is defined in software as these functions:

```
// Execute a single instruction in the given X16 machine. Update
// memory and registers as required. PC is advanced as appropriate.
// Return 0 on success, or -1 if an error or HALT is encountered.
int execute_instruction(x16_t* machine);

// Update condition code in R_COND based on result in the given register
void update_cond(x16_t* machine, reg_t reg);
```

The function `execute_instruction` executes one instruction in the emulator. The function is called repeatedly from the main loop until it returns -1 on error or a HALT instruction is encountered.

## Main Program

The main program usage is as follows:

```
USAGE: ./x16 [-l] [objectfile]
```

With the `-l` argument, the program turns on logging. This prints out each instruction as the emulator fetches it. It is very useful for debugging. With no objectfile argument, the program opens and reads the X16 object file named `"a.obj"` by default. Otherwise it opens and reads the given object file into memory.

```
int main(int argc, char** argv) {
    int ch;
    while ((ch = getopt(argc, argv, "l")) != -1) {
        switch (ch) {
        case 'l':
            LOG = 1;
            break;

        default:
            usage();
        }
    }
    argc -= optind;
    argv += optind;


    char* filename = "a.obj";
    if (argc > 1) {
        usage();
    } else if (argc == 1) {
        filename = argv[0];
    }

    // Initialize machine
    x16_t* machine = x16_create();

    // Read the image file into memory
    if (read_image(machine, filename) != 0) {
        fprintf(stderr, "Failed to read image: %s\n", filename);
        exit(1);
    }

    // Set up signal handler to clean up TTY state on SIGINT
```

```
    signal(SIGINT, handle_interrupt);

    // Disable so we can read keystrokes without newline
    disable_input_buffering();

    // Execute the emulation till we see a halt or some error occurs
    for (;;) {
        if (LOG) {
            x16_print(machine);
        }
        if (execute_instruction(machine) != 0) {
            break;
        }
    }

    // Restore TTY state
    restore_input_buffering();

    x16_free(machine);
}
```

The main loop executes the machine by calling `execute_instruction` until it sees an error or a HALT instruction. The disable and restore input buffering methods control Unix IO to the console. It allows the X16 emulator to read a single character without the user having to press enter. The signal handler restores standard input buffering in case the user types control-C to interrupt the emulation. Without it, the console may be left in a weird state.[2]

## Image Files

The image file is read into memory with the `read_image` function. **All words in X16 are stored on disk in Big Endian format**, hence we need to swap from network byte order[3] to host byte order.

The first word in the object file is the location in memory to read the data into. Since we are dealing with user programs for X16, this value is always 0x3000. Afterwards, we read the rest of the object file into the memory array at the right location. Since each word is also stored in Big Endian format, we convert it to host format.

```
// Read Image File. Return 0 on success or -1 for failure
static int read_image_file(x16_t* machine, FILE* fp) {
```

---

[2] If the terminal is in a weird state, entering the command `"reset"` will usually restore the console to the right default state.

[3] Network byte order is defined to always be big endian.

```
    // The origin tells us where in memory to place the image
    uint16_t origin;
    if (fread(&origin, sizeof(origin), 1, fp) <= 0) {
        return -1;
    }
    // Swap to host format
    origin = ntohs(origin);

    // we know the maximum file size so we only need one fread
    uint16_t max_read = UINT16_MAX - origin;
    uint16_t* p = x16_memory(machine, origin);
    size_t read = fread(p, sizeof(uint16_t), max_read, fp);
    if (read <= 0) {
        return -1;    // nothing read, or some error in fread
    }

    // swap each 16 bit value to host format
    while (read-- > 0) {
        *p = ntohs(*p);
        ++p;
    }

    return 0;
}
```

# Your Tasks (250 total points possible)

Your task is to **implement the emulator** given the skeleton code, and **implement an assembler** that turns assembly language for the X16 into a X16 object file that the emulator can execute.

## Emulator Tasks

### Task 1. Bit Handling (5 points)

The first task is bit handling. In **bits.h** and **bits.c**, you're given these functions. They are useful for handling bit fields within instructions, and you'll find them handy in later tasks.

```
// Get the nth bit in the number
uint16_t getbit(uint16_t number, int n);

// Get bits at location n that are the given number of bits wide
uint16_t getbits(uint16_t number, int n, int wide);
```

```
// Return number with the nth bit set to 1
uint16_t setbit(uint16_t number, int n);

// Return number with the nth bit cleared to 0
uint16_t clearbit(uint16_t number, int n);

// Sign extend a number with the given bit count to 16 bits
uint16_t sign_extend(uint16_t x, int bit_count);
```

You're given these helper functions that will help you determine if a 16 bit number is positive or negative.

```
// True if the number is positive
bool is_positive(uint16_t number);

// True if the number is negative
bool is_negative(uint16_t number);
```

For sign_extend, the most significant bit of A is replicated as many times as necessary to extend A to 16 bits. For example, if A = 110000, then sign_extend(A) = 1111 1111 1111 0000.

To ensure that you have bit handling working, you can test with

```
make test-bits
```

## Task 2. Instruction Decoding (2 points)

In **instruction.h** and **instruction.c** you are given these functions.

```
// Get opcode from the instruction. The opcode is the highest 4 bits
// of the instruction.
opcode_t getopcode(uint16_t instruction);

// Get the immediate bit (5th bit)
uint16_t getimmediate(uint16_t instruction);
```

Using the bit handling functions from earlier, implement these two functions.

To ensure that these work correctly, you can test with

```
make test-instruction
```

## Task 3. Control Unit (10 points for each instruction)

Your next task is to implement the control logic that handles each instruction. Flesh out `execute_instruction` in control.c.

Start with the ADD instruction. Test that your implementation is correct with

```
make test-control-add
```

There are 12 instructions, and you can test each instruction with:

```
make test-control-add
make test-control-and
make test-control-br
make test-control-jmp
make test-control-jsr
make test-control-ld
make test-control-ldi
make test-control-lea
make test-control-ldr
make test-control-st
make test-control-sti
make test-control-str
```

You don't have to write the code for handling TRAPs. That is written for you in the skeleton code.

### How to Tackle This

The best way to attack this is to do **one instruction at a time**.

Your plan of attack should be:

- Look at the X16 ISA Reference for the instruction you want to implement. Understand how the instruction is supposed to work.
- Implement the instruction and run the corresponding test.
- You can also look at the test code in the test directory to see how each instruction is tested.

### Game On

**Once all the instruction tests pass, you should have a fully functioning X16 emulator**. You will have reached the point where you should be able to play a few games written for X16.

Compile the x16 emulator with

```
make x16
```

Then load the game Rogue with:

```
./x16 rogue.obj
```

You should also be able to play 2048 with:

```
./x16 2048.obj
```

## Assembler Tasks (100 points)

Your last task is to write an assembler for the X16. The assembler is called **xas**. Your code should be in **xas.c**. The usage is

```
Usage: ./xas filename
```

It accepts exactly one argument, the name of the assembler file to process, and produces the output file **a.obj**. The output file is always set to load at 0x3000, the start of the user address space for X16. This is always the first 16-bit value in an object file. After that, there are a series of 16-bit values corresponding to instructions and 16-bit data values for the object file.

### Return Value

The program returns 0 on success, 1 if there was no file specified, or 2 if any error is encountered, e.g. if the input assembler is badly formed. There is no requirement to output a useful error message, though of course it's useful for assembler writers to see where the error is.

### Example Assembler Program

An example assembler program is given below. Our assembler uses syntax inspired by GNU AT&T assembler.

```
        # this is a comment
start:
        add %r1, %r0, $10   # This is a comment

start1:
        ld  %r0, star
        putc
        add %r1, %r1, $-1
        brz stop
```

```
        jsr start1

stop:
        halt

star:
        val $42
newline:
        val $10
```

The image file produced by the example will have a series of 16-bit words stored in Big Endian format. The first word is 0x3000, the location to load the program. It is always that value, since our assembler only produces user programs. Then there are 7 instructions for (add, ..., halt) followed by two 16-bit data values of 42 and 10 respectively.

## Assembly Language

The features of the assembly language are:

- # starts a comment
- Labels end with a colon and are always on a line by themselves
- Instructions are lower case
- Register names have a % prepended
- Values have a $ prepended
- To insert a 16-bit value into memory, the pseudo instruction is `val`. For example, the last line of the example above inserts a value of `0x10` (ASCII newline) into the memory location specified by the label "newline".
- IO traps have these pseudo instructions:
    - `getc` for TRAP_GETC
    - `putc` for TRAP_OUT
    - `puts` for TRAP_PUTS
    - `enter` for TRAP_IN
    - `putsp` for TRAP_PUTSP
    - `halt` for TRAP_HALT

## Emitting Instructions

You're given a whole bunch of `emit_*` functions in instruction.h you can use. For example, to emit a jsr instruction, just call `emit_jsr(offset)` where the offset is a 16-bit value you want to use. This returns a 16-bit value you can write to the output file (don't forget to save it in Big Endian format).

## Dealing with Labels

Notice that your assembler needs to deal with labels. As you parse each line of the assembler program, some labels refer to locations that you haven't encountered yet. The only way to deal with this is to pass over the data twice in what is called **two pass assembly**.

The first pass collects instructions and labels with their locations. Once the first pass is complete, you will know where in memory every label refers to. In the second pass, you go through the list of instructions again and update the offsets for each instruction to the correct value.

You're free to approach this part in any way you wish. Typical assemblers usually write the output file with zeros for all the offsets, then go back and fill in the zeros with the right values. In our case, it might be simpler to read everything into an in-memory representation, then fix everything up in a second pass through memory.

## Building and Testing

Build your assembler with

```
make xas
```

Test your assembler with

```
make test-xas
```

Try running your own assembler programs with your own emulator.

# Disassembler

To help you with writing the assembler, I have provided a disassembler called **xod**. You can build xod with

```
make xod
```

The usage is

```
Usage: ./xod [filename]
```

If no file name is specified, xod reads "a.obj", otherwise the filename is opened and dumped. Here is the result when xod is run against the object file produced by compiling the example test program above.

```
Origin: 0x3000
0x3000: 0001 0010 0010 1010 : add    %r1, %r0, $10
```

```
0x3001: 0010 0000 0000 0101 : ld     %r0, $5
0x3002: 1111 0000 0010 0001 : putc
0x3003: 0001 0010 0111 1111 : add    %r1, %r1, $-1
0x3004: 0000 0100 0000 0001 : brn    1
0x3005: 0100 1111 1111 1011 : jsr    $-5
0x3006: 1111 0000 0010 0101 : halt
0x3007: 0000 0000 0010 1010 : br     $42
0x3008: 0000 0000 0000 1010 : br     $10
```

The first line prints out the origin where the program is loaded. The lines after that represent each 16-bit instruction or value. The first column is the hexadecimal address. The next 4 columns are the binary representation of the instruction. Then the disassembly of the instructions follows. Notice that the last two values are represented as `br` instructions and are not decoded correctly. The disassembler doesn't know that these are values, so be careful when using xod.

## Memory Leaks (13 points)

There are 5 points for passing a valgrind check for no memory leaks in X16.

There are 8 points for passing a valgrind check for no memory leaks in XAS.

## Style Check (10 points)

There are 10 points for C-style.

# X16 Instruction Set Architecture

## Overview

The Instruction Set Architecture (ISA) of X16 is defined as follows[4]:

### Memory address space

16 bits, corresponding to $2^{16}$ locations, each containing one word (16 bits). Addresses are numbered from 0 (i.e, x0000) to 65,535 (i.e., xFFFF). Addresses are used to identify memory locations and memory-mapped I/O device registers. Certain regions of memory are reserved for special uses, as described in Figure 1.

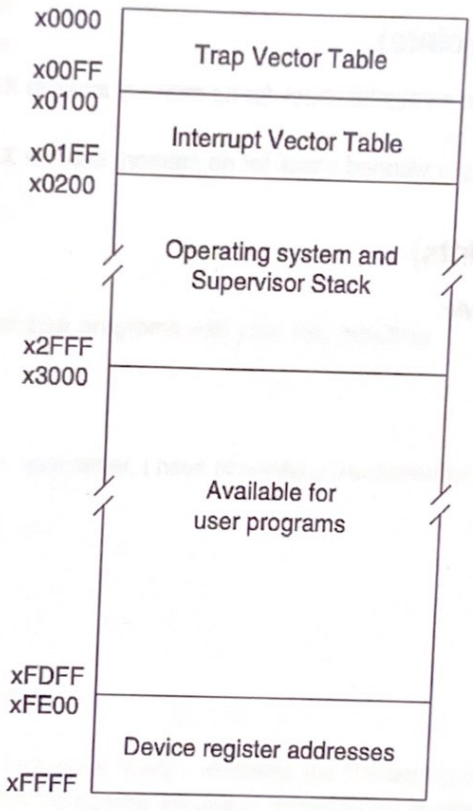



Figure 1. Memory map of X16. The interrupt vector table and OS region is not used in X16.

[4] X16 ISA is a simplified ISA that elides supervisor, interrupt and stack modes. Credit goes to Yale Patt at UTA and Sanjay Patel at UIUC for the original concept.

## Bit numbering

Bits of all quantities are numbered, from right to left, starting with bit 0. The leftmost bit of the contents of a memory location is bit 15.

## Instructions

Instructions are 16 bits wide. Bits [15:12] specify the opcode (operation to be performed), bits [11:0] provide further information that is needed to execute the instruction. The specific operation of each instruction is described in the following section.

## Illegal opcode exception

Bits [15:12] = 1101 has not been specified. If an instruction contains 1101 in bits [15:12], an illegal opcode exception occurs.

## Program counter

A 16-bit register containing the address of the next instruction to be processed.

## General purpose registers

Eight 16-bit registers, numbered from 000 to 111.

## Condition codes

Three 1-bit registers: N (negative), Z (zero), and P (positive). Load instructions (LD, LDI, LDR, and LEA) and operate instructions (ADD, AND, and NOT) each load a result into one of the eight general purpose registers. The condition codes are set, based on whether that result, taken as a 16-bit 2's complement integer, is negative (N = 1; Z, P = 0), zero (Z = 1; N, P = 0), or positive (P = 1; N, Z = 0).

All other X16 instructions leave the condition codes unchanged.

## Memory-mapped I/O

Input and output are handled by load/store (LDI/STI, LDR/STR) instructions using memory addresses to designate each I/O device register. Addresses 0xFE00 through 0xFFFF have been allocated to represent the addresses of I/O devices. See Figure 1. The memory mapped IO section lists each of the relevant device registers that have been identified for the X16 thus far, along with their corresponding assigned addresses from the memory address space.

## Interrupt processing

Interrupt processing is not done on X16.

# Notation

| Notation | Meaning |
|---|---|
| xNumber | The number in hexadecimal notation. |
| #Number | The number in decimal notation. |
| A[l:r] | The field delimited by bit [l] on the left and bit [r] on the right, of the datum A. For example, if PC contains 0011001100111111, then PC[15:9] is 0011001. PC[2:2] is 1. If l and r are the same bit number, the notation is usually abbreviated PC[2]. |
| BaseR | Base Register; one of R0..R7, used in conjunction with a six-bit offset to compute Base+offset addresses. |
| DR | Destination Register; one of R0..R7, which specifies which register the result of an instruction should be written to. |
| imm5 | A 5-bit immediate value; bits [4:0] of an instruction when used as a literal (immediate) value. Taken as a 5-bit, 2's complement integer, it is sign-extended to 16 bits before it is used. Range: −16..15. |
| LABEL | An assembly language construct that identifies a location symbolically (i.e., by means of a name, rather than its 16-bit address). |
| mem[address] | Denotes the contents of memory at the given address. |
| offset6 | A 6-bit value; bits [5:0] of an instruction; used with the Base+offset addressing mode.<br>Bits [5:0] are taken as a 6-bit signed 2's complement integer, sign-extended to 16 bits and then added to the Base Register to form an address. Range: −32..31. |
| PC | Program Counter; 16-bit register that contains the memory address of the next instruction to be fetched. For example, during execution of the instruction at address A, the PC contains address A + 1, indicating the next instruction is contained in A + 1. |
| PCoffset9 | A 9-bit value; bits [8:0] of an instruction; used with the PC+offset addressing mode. Bits [8:0] are taken as a 9-bit signed 2's complement integer, sign-extended to 16 bits and then added to the incremented PC to form an address. Range −256..255. |
| PCoffset11 | An 11-bit value; bits [10:0] of an instruction; used with the JSR opcode to compute the target address of a subroutine call. |

| | |
|---|---|
| | Bits [10:0] are taken as an 11-bit 2's complement integer, sign-extended to 16 bits and then added to the incremented PC to form the target address. Range −1024..1023. |
| setcc() | Indicates that condition codes N, Z, and P are set based on the value of the result written to DR. If the value is negative, N = 1, Z = 0, P = 0. If the value is zero, N = 0, Z = 1, P = 0. If the value is positive, N = 0, Z = 0, P = 1. |
| SEXT(A) | Sign-extend A. The most significant bit of A is replicated as many times as necessary to extend A to 16 bits. For example, if A = 110000, then SEXT(A) = 1111 1111 1111 0000. |
| SR, SR1, SR2 | Source Register; one of R0..R7 which specifies the register from which a source operand is obtained. |
| trapvect8 | An 8-bit value; bits [7:0] of an instruction; used with the TRAP opcode to determine the starting address of a trap service routine. Bits [7:0] are taken as an unsigned integer and zero-extended to 16 bits. This is the address of the memory location containing the starting address of the corresponding service routine. Range 0..255. |
| ZEXT(A) | Zero-extend A. Zeros are appended to the leftmost bit of A to extend it to 16 bits. For example, if A = 110000, then ZEXT(A) = 0000 0000 0011 0000. |

# The Instruction Set

Each 16-bit instruction consists of an opcode (bits[15:12]) plus 12 additional bits to specify the other information that is needed to carry out the work of that instruction. Figure 2 summarizes the 15 different opcodes in X16 and the specification of the remaining bits of each instruction. The 16th 4-bit opcode is not specified, but is reserved for future use. In the following pages, the instructions will be described in greater detail.

For each instruction, we show the assembly language representation, the format of the 16-bit instruction, the operation of the instruction, an English-language description of its operation, and one or more examples of the instruction. Where relevant, additional notes about the instruction are also provided.

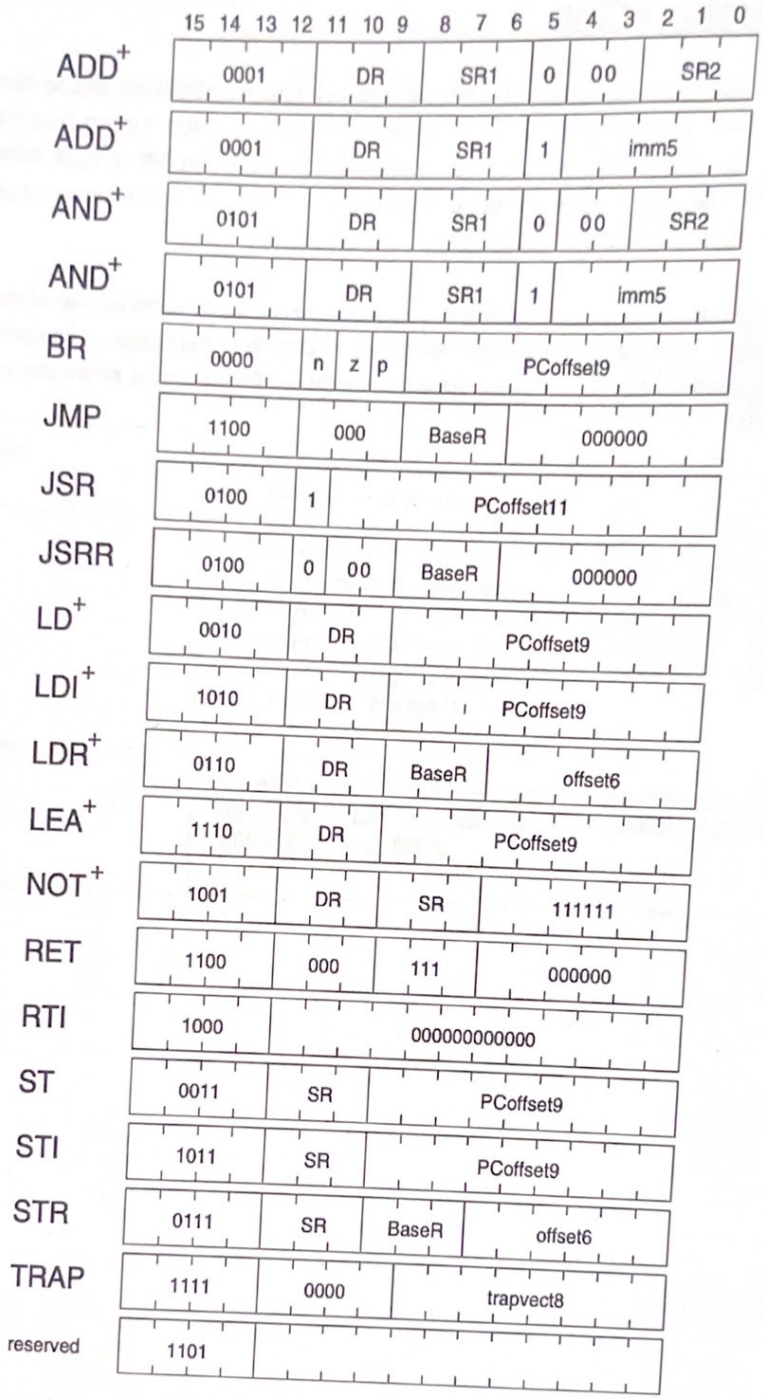


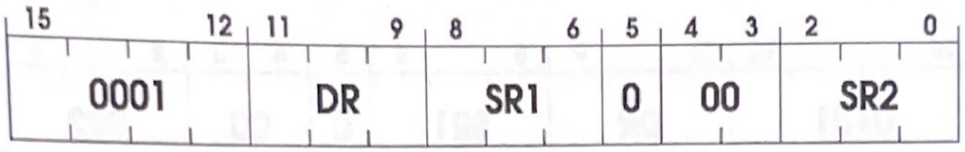

Figure 2. Format of the entire X16 instruction set. + indicates instructions that modify condition codes. RET and RTI are not used in X16.

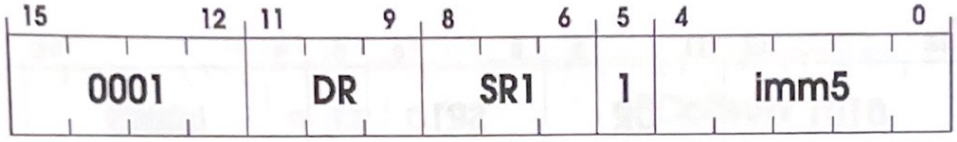
# ADD - Addition

## Assembler Formats

```
ADD DR, SR1, SR2
ADD DR, SR1, imm5
```

## Encodings




## Operation

```
if (bit[5] == 0)
      DR = SR1 + SR2;
else
      DR = SR1 + SEXT(imm5);
setcc();
```

## Description

If bit [5] is 0, the second source operand is obtained from SR2. If bit [5] is 1, the second source operand is obtained by sign-extending the imm5 field to 16 bits.

In both cases, the second source operand is added to the contents of SR1 and the result stored in DR. The condition codes are set, based on whether the result is negative, zero, or positive.

## Examples

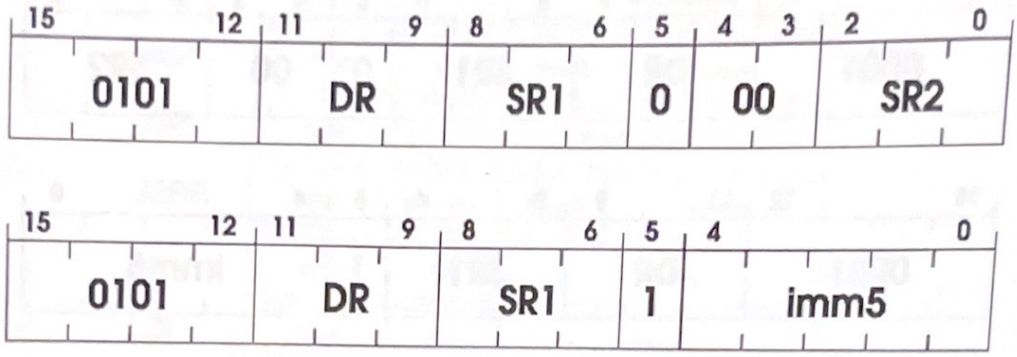
```
ADD R2, R3, R4      ; R2 ← R3 + R4
ADD R2, R3, #7      ; R2 ← R3 + 7
```

# AND - Bit-wise Logical AND

## Assembler Formats

```
AND DR, SR1, SR2
AND DR, SR1, imm5
```

## Encodings




## Operation

```
if (bit[5] == 0)
      DR = SR1 AND SR2;
else
      DR = SR1 AND SEXT(imm5);
setcc();
```

## Description

If bit [5] is 0, the second source operand is obtained from SR2. If bit [5] is 1, the second source operand is obtained by sign-extending the imm5 field to 16
bits. In either case, the second source operand and the contents of SR1 are bit-wise ANDed, and the result stored in DR. The condition codes are set, based on whether the binary value produced, taken as a 2's complement integer, is negative, zero, or positive.

## Examples

```
AND R2, R3, R4          ;R2 ← R3 AND R4
AND R2, R3, #7          ;R2 ← R3 AND 7
```