Chapter 2: Processor Architecture

What is involved in processor design?

- Think of hardware resoures as the alphabet of a language
- Instruction Set (ISA) is akin to the words in a language
    - Contract between the software and the actual hardware implementation

Common High Level Language Feature Set

There are a few features we need to consider

1. Expressions and assignment statements
2. High-level data abstractions
3. Conditional statements and loops
4. Procedure Calls

Expressions and assignment statements

- Arithmetic and logic operations
- Consider the size and location of operands

High Level Data Abstraction

- Aggregate simple variables into structures or records

Conditional Statements and Loops

- Change the sequential flow of the execution of a program

Procedure Calls

- Remember the state of a program before a procedure call
- Pass the parameters
- Receive result from parameters

Expressions and Assignment Statements

Consider the high level arithmetic and logical expressions

```java
a = b + c;
d = e - f;
x = y & z;
```

In the ISA, it becomes

```
add a, b, c
sub d, e, f
and x, y, z
```

The above instructions are also called *binary* instructions since they work on two operands to produce a result

- Also called three operand instructions since there are 2 source operands and one destination oeprand

Where to keep the operands?

If the operands of an instruction are in the register, then they are much quicker to access than memory

There is also the problem of the *addressability* of operands

- If you have a very large toolbox and your project requires you to pretty much use all the tools in your toolbox, you don't want to store everything in the toolbox, return it, and then fetch it again when you need it
  - Instead, you re-use the space in the tool tray for different tools
- As the size of memory grows, the size of a memory address (number of bits needed to uniquely name a location) also grows
- If an instruction has to name three memory operands, then that also increases the size of each individual instruction

Registers solve this problem by storing these operands in the register

- Registers have a set number of bits

Thus, our ISA instructions look like this now

```
add r1, r2, r3
sub r4, r5, r6
and r7, r8, r9
```

Sometimes, we need **immediate values**

```
addi r1, r2, imm;
```

To solve this problem, we introduce the concept of ***addressing mode***, which refers to how the operands are specified in an instruction

In the case above, the addressing mode used is called ***register addressing*** since the operands are in registers

We modify our instructions to look something like this now

```
a = b + c



ld r2, b
ld r3, c
add r1, r2, r3
st r1, a
```

Here is an example of why you would want to use registers

```
d = a * b + a * c + a + b + c
```

Once `a`, `b`, and `c` are brought from the memory into registers they can be reused several times in just this one expression evaluation

How do we specify a memory address in an instruction?

To mitigate needing the entire memory address of an operand in each instruction, we introduce the **base + offset** mode

Mnemonically, this is usually `ld r2, offset(rb); r2 <- MEMORY[rb + offset]`

- `rb` = memory address of variable `b` or any one of the registers

```
LW    Rx, Ry, OFFSET
```

How wide should each operand be?

This question is referred to as the *granularity* or *precision* of the operands

- Ex: char, short, long, int

Word precision refers to the maximum precision the architecture can support in hardware for arithmetic/logic operations

Addressability refers to smallest precision operand that can individually addressed in memory

- Byte-addressable = smallest precision that can be addressed is a byte

If we have `0x11223344`, then in memory this will look like

And we can individually address each of these bytes (remember 2 hex characters = 1 byte)

Endianness

For a word at location 100 containing the values `0x11223344`

Big Endian

11	22	33	44
100	101	102	103

Little Endian

11	22	33	44
103	102	101	100

Usually in high level languages, this doesn't matter as long as you use a data type the way it was declared. However...

```c
int i = 0x11223344;
char *c;

c = (char *) &i;
printf("endian: i = %x; c =  %x\n", i, *c);
```

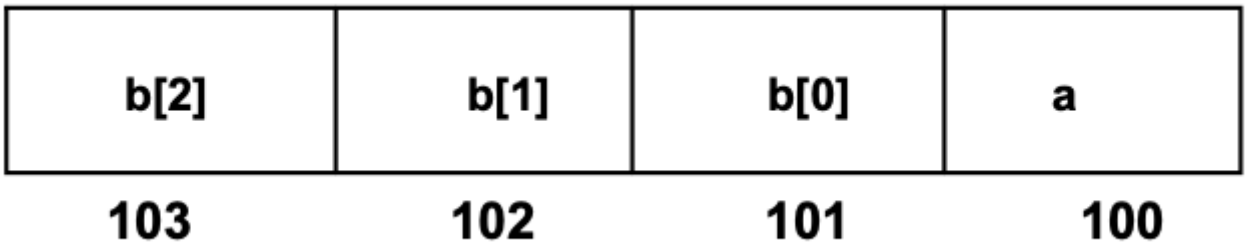
In a big endian machine, c would be printed as `0x11`, while in a little endian machine, c would be printed out as `0x44`

Packing Operands

A compiler may try to pack operands of a program to conserve space

```c
struct {
    char a;
    char b[3];
}
```

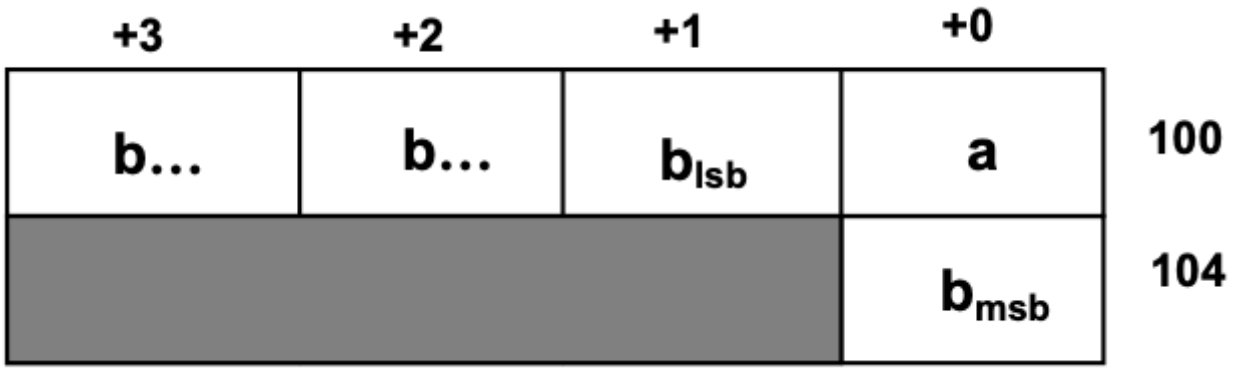
This might be optimized and packed as



However, packing might not always be the right idea.

```c
struct {
    char a;
    int b;
}
```

If we try to pack



The problem with this layout is that `b` is an int and it starts at address 101 and ends at address 104. To load `b`, we have to bring in two words, which is inefficient.

Architectures typically require *alignment restriction* of word operands to word addresses

Therefore, the better way to layout the structure in memory is

High Level Data Abstractions

Structures

```c
struct {
    int a;
    char c;
```

```
    int d;
}
```

If the base address of the structure is in some register, then accessing any field within the struct can be accomplished using base + offset.

Arrays

Since arrays are typically used in loops, the offset to the base register is not fixed. Some architectures will introduce a **base + index** addressing mode that allows the effective address to be computed as the sum of the contents of two registers.

```c
a[j] = a[j] + 1;
```

- `j` might not be fixed
- Store `j` in a register, use base + offset

Conditional Statements and Loops

```c
if(j == k) goto L1;
a = b + c;

L1: a = a + 1;
```

Consider the above statements. To complete the `if` statement, we need

1. Evaluation of the predicate `j==k`
2. If the predicate evalutes to TRUE, then it changes the flow of control from going to the next sequential instruction to the target L1

We introduce a conditional branch

```
beq r1, r2, L1
```

The effect of this instruction is

1. Compare r1 and r2
2. If they are equal, then the next isntruction to be executed is at `PC + offset*`

3. If they are unequal, then the next instruction is the next one textually following beq `*` offset is adjustedfor implementation specifics

In the case of an else, we don't need to introduce anything new, just use an unconditional jump

```
j   rtarget
```

We can technically accomplish this with just `BEQ R1, R1, offset`, but there is a limit to the offset. That is why we introduce a new unconditional jump instruction.

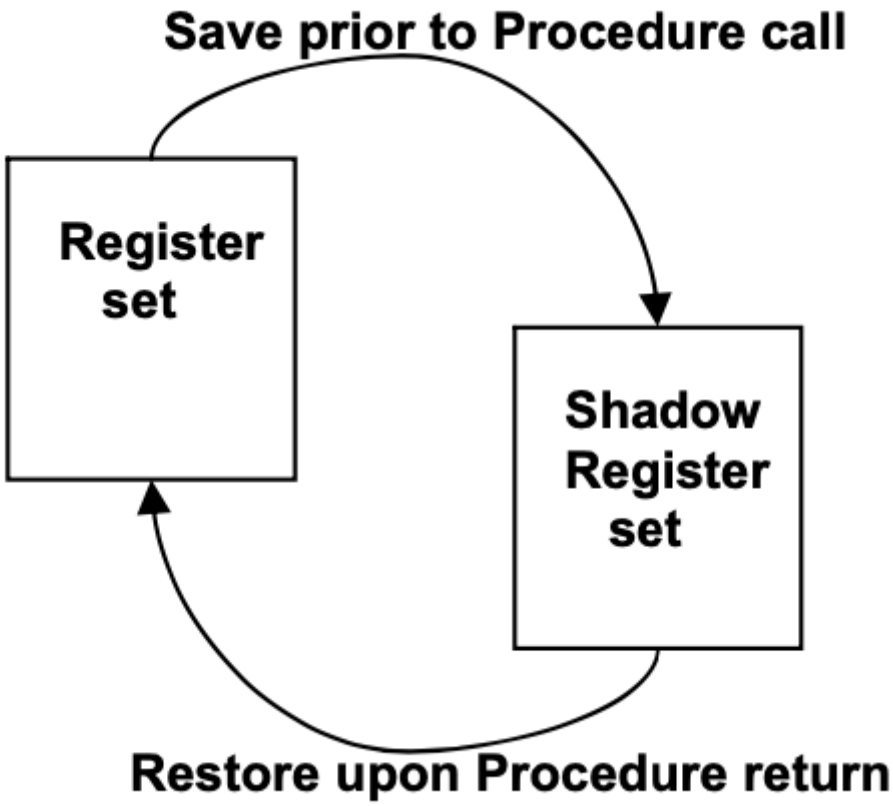
Function Calls

Let's enumerate the steps in compiling a procedure call

1. Ensure that the state of the caller (i.e. processor registers used by the caller) is preserved for resumption upon return from the procedure call
2. Pass the actual parameters to the callee
3. Remember the return address
4. Transfer control to callee
5. Allocate space for callee's local variables
6. Receive the return values from the callee and give them to the caller
7. Return to the point of call

State of the Caller

We need to save the state. One of the ways we can do this is with **shadow registers**



Note that this is a hardware solution, so we assume that we hope we won't run into issues with needing very many shadow registers.

The other way we can handle this is through software: the **stack**

- Compiler maintains a pointer to the stack for saving and restoring state called the **stack pointer**
- Problem is we have to move data back and forth between processor registers and memory on each procedure call/return

Well then, which registers should you save?

Story of two lazy roommates

- They want to get away with as little house chores as possible
- They have a frying pan for common use and a set of their own dishes
  - Dishes that are their own they never have to clean
  - If they use other person's dishes then they have to clean it after every use
  - There is no guarantee that the frying pan will be clean; it is up to each person to clean and use it if they need it

Similarly, the caller gets a set of the registers that are its own (***s*** registers). The caller can use them any which way it wants. The callee, if it wants to use those registers, has to restore them. Then, there are a subset of registers (***t*** registers) that are commong to both

Remaining chores with procedure calling

1. Paramter passing
2. Remember return address
3. Transfer control to callee
4. Space for callee's local variables
5. Return values
6. Return to the point of call

Software Convention

- Registers s0-s2 are the caller’s s registers
- Registers t0-t2 are the temporary registers
- Registers a0-a2 are the parameter passing registers
- Register v0 is used for return value
- Register ra is used for return address
- Register at is used for target address
- Register sp is used as a stack pointer

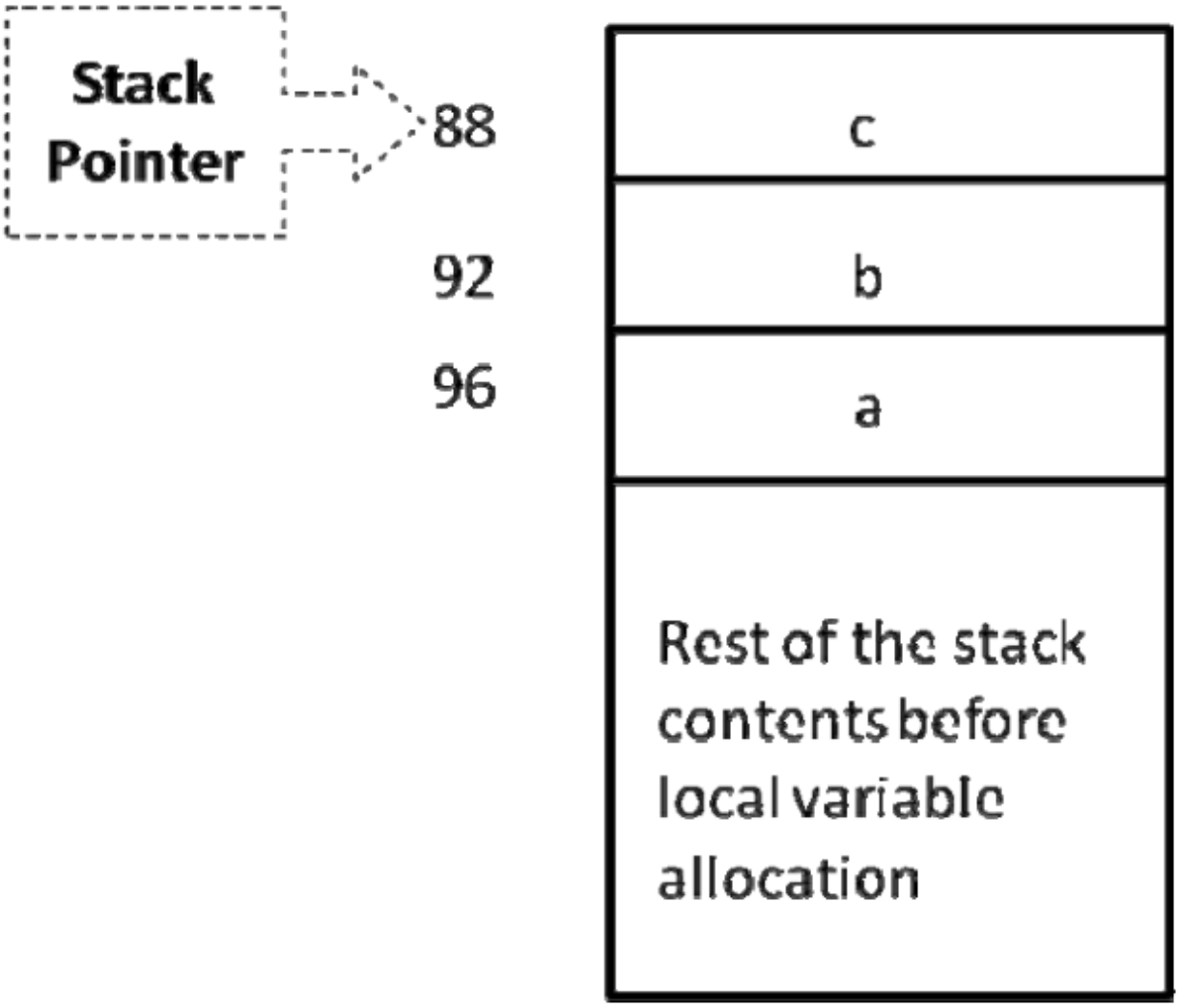
The **activation record** is the portion of the stack that is relevant to the currently executing procedure

Frame Pointer

During execution of a program, it is obviously essential to be able to locate all of the items that are stored on the stack by the program.

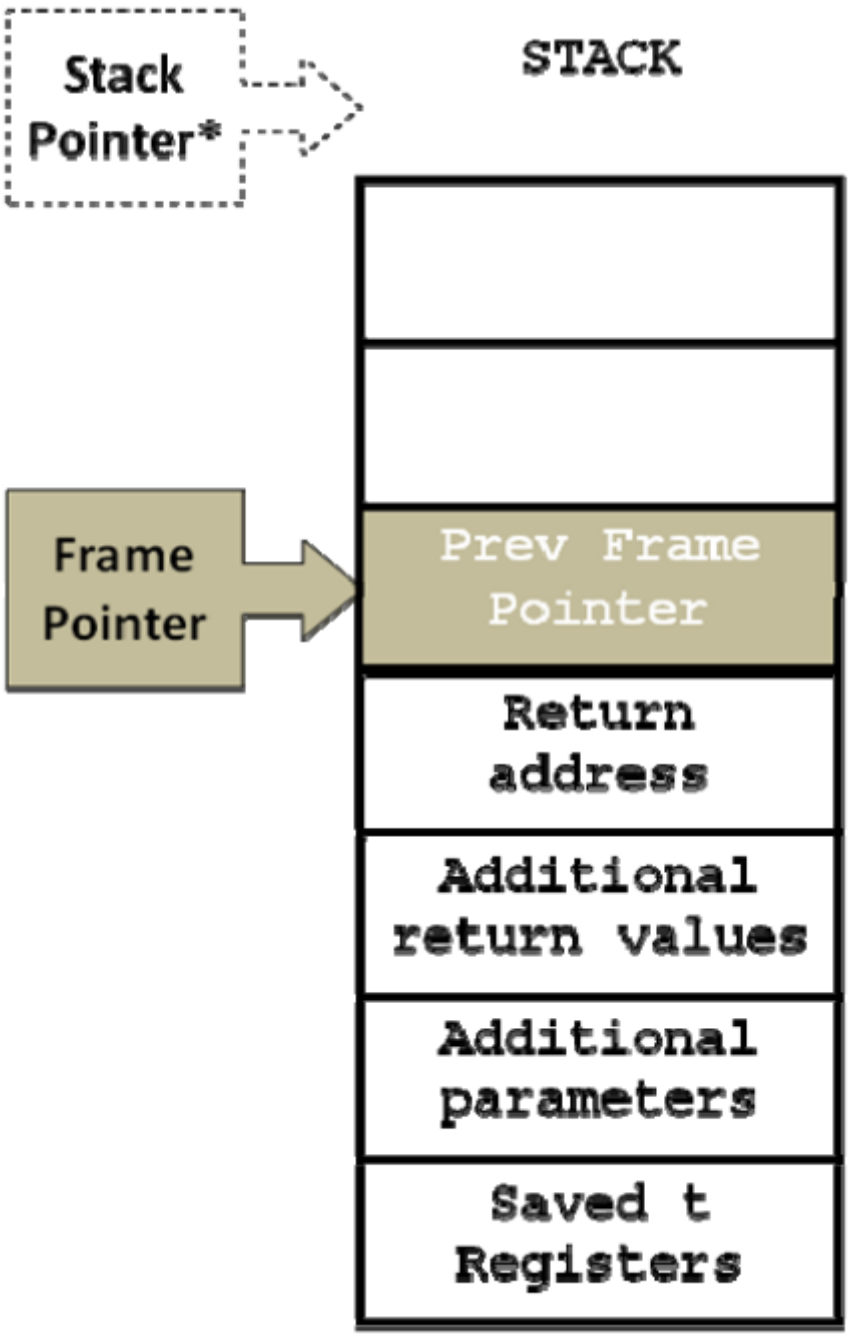
To keep up with dynamic stack allocation, we designate a register as a ***frame pointer***

- Contains address of a known point in the activation record for the function and never changes during the execution of the function



- What if c is allocated conditionally?
  - We need the load/store the variable `a` and `b` accurately
  - If `c` is allocated, then a is just offset 8
  - But what if `c` did not get allocated?

The frame pointer contains the first address on the stack the pertains to the activation record of the called procedure



Instruction Format

1. Zero operand instructions
2. One operand instructions
3. Two operand instructions
4. Three operand instructions

Instructions can be grouped into two categories

Instructions are of the same length

- Pros
  - Simplifies implementation
  - Interpretation of the fields of the instruction can start speculatively as soon as the instruction is available
- Cons
  - Potential wasted space
  - Additional glue logic (decoders and mux)
  - Instruction-set designer is more limited

Example of same length instruction architecture: MIPS

Instructions are of variable length

- Pros
  - No wasted space
  - Instruction set designer not contrained by limited sizes
  - Choose different sizes and encoding for opcodes, addressing modes, and operands
- Cons
  - Complicates implementation since length of the isntruction can be discerned only after decoding opcode

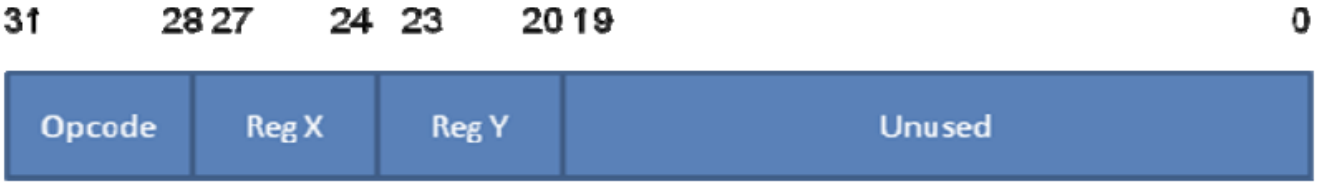
Example of variable length ISA: DEC VAX 11 and Intel x86

LC-2200 Instruction Format

R-type instructions (add, nand)

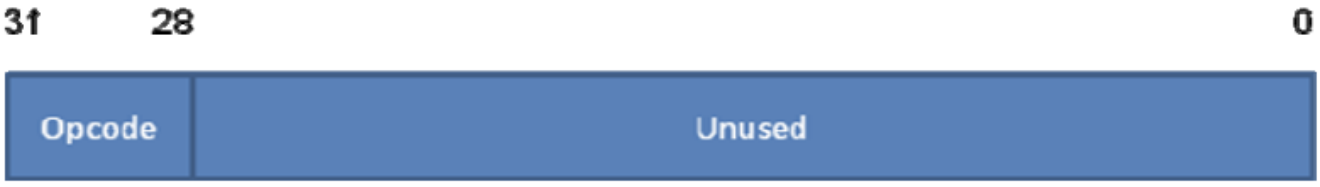
I-type instructions (addi, lw, sw, beq)

J-type instruction (jalr)



- reg X = target of jump
- reg Y = link register

O-type instructions (halt)



LC-220 Register Set

Reg #	Name	Use	callee-save?
0	$zero	always zero (by hardware)	n.a.
1	$at	reserved for assembler	n.a.
2	$v0	return value	No
3	$a0	argument	No
4	$a1	argument	No
5	$a2	argument	No
6	$t0	Temporary	No
7	$t1	Temporary	No
8	$t2	Temporary	No
9	$s0	Saved register	YES
10	$s1	Saved register	YES
11	$s2	Saved register	YES
12	$k0	reserved for OS/traps	n.a.
13	$sp	Stack pointer	No
14	$fp	Frame pointer	YES
15	$ra	return address	No