Memory Organisation

Last time,

Register file in processor, with 32 registers

The program has access to a larger memory.

1. Let's say prog has 400 instrs. Each instr takes 4 bytes

2. The base point from where the variables start getting stored in "text" (insts) is 1600 bytes. This region is called "gp".

3. In this example, 4 gp = 1600.

4. Points to the region stored in a register called "gp".

5. The first set of variables are the global variables. (The very first vars you declare before you execute anything).

6. Static data (globals)

7. Text (insts)

8. The compiler knows all these global vars (the variables that are declared at the start).
9. These variables get allocated beforehand, in the global region. (before runtime, during execution)

10. From that point on, variables are allocated as per need.

STACK

- This is the stack, which is a part of the heap region of memory.

- At compile time, the compiler does NOT know what gets placed here.
- These are determined at runtime.
- Stores procedure variables when you invoke the procedure, e.g.

11. When you invoke main(), vars only accessible to main, (all variables defined in the file) are declared here.
main() \rightarrow find()

Those its variables get declared here.

On top of the stack growing downwards.

Once the fn finishes they get deallocated.

In this way, the stack shoves up & down.

**Dynamic Data**

Program can dynamically allocate memory where to define explicitly - i.e. defining how much memory to allocate to allocate.

E.g. in C, we use malloc().

All those vars are stored in heap.

The heap starts where the global variables storage end.
Stack

* Heap address
  - Knob memory
  - Upper address

Base address & offset

\[ a = b + c \]

Example code:
```c
int a, b, c, d[10];
```

Cells of program had 8 bit instruction.

```
1, 2, 3
```

Code neighbors each other space each other.
So global variable starts at address 1000,

so, I do, add 1000 to global pointer.

\[ \text{addi } gp, \ zzero, 1000 \]

\[ \text{loew } s51, 4(gp) \]
\[ \text{stw } s52, 8(gp) \]
\[ \text{add } s51, s52, s53 \]
\[ \text{sw } s51, gp \]
\[ \text{addi } s54, gp, 12 \]
\[ \text{lw } s10, 8(s54) \] # get d127
\[ \text{add } s10, s10, s51 \]
\[ \text{sw } s10, 12(s54) \]
Instruction Formats (Have 16th format stride)

2. Broad classes of instructions

R-type

Last 6 bits denotes the variation in adding: Adding a byte, 2 bytes, or a word?

The 5 bits shift is used for operation instruction like su, sol

By which bits are shifted to the left or right.

These bits tell us how much this shift is.

I-type

e.g. Immediate,

(u, s, u)

5 bits → 1st reg. operand

16 bits → next reg. operand

5 bits → stores the offsets.
beg $s1, $s2, $1

L1:

breq

srl $t1, $t0, 1

set $t0 to 1 if $f1 < $f2

can be used with a following branch instruction which checks the result of srl

2nd kind: Unconditional branch:

jra jump to address in a register
beg $s1, $s2, $t1

L1:

breq

see slt \rightarrow \text{not itself a branch}
(set on less than)

\( \oplus \) set a certain value based on a comparison of 2 registers

e.g. slt $t0, $f1, $f2
    set $t0 to 1 if $f1 < $f2

\text{can be used with a followup branch instruction which checks the result of slt and proceeds accordingly}

2nd kind: Unconditional branch
\text{jr} \rightarrow \text{jump to address in a register}
Task (Code)

if (i == j)
    \( f = g + h; \)
else
    \( f = g - h; \)

[Assembly]
Say: $51 \& $52
Have i \& \sqrt{j}.

[explain why]

while (save \( \text{%eax} \)
    \( = k \))
    \( i' = 1; \)

Say: b, k, same (base) are
and $53, $55, $56

Ans:

A label can also be an instruction (an empty instruction) assembler/computer will read it as a jump location.

loop:

P: T: O
Loop:
```
sll $t1, $s3, 2
add $t1, $t1, $s6
lw $t0, 0($t1)
bne $t0, $s5, Exit
add $s3, $s3, 1
```

Exit:
```
j Loop
```

1. Jump to the top of the loop
2. Get the value
3. Set in $s6
4. Shift i by 2 to the left

```
save[j] -> is + in $s6
```
```
save[i] -> $s6 + 4*i
```
```
save[i] -> $s6 + 4*i
```
```
```