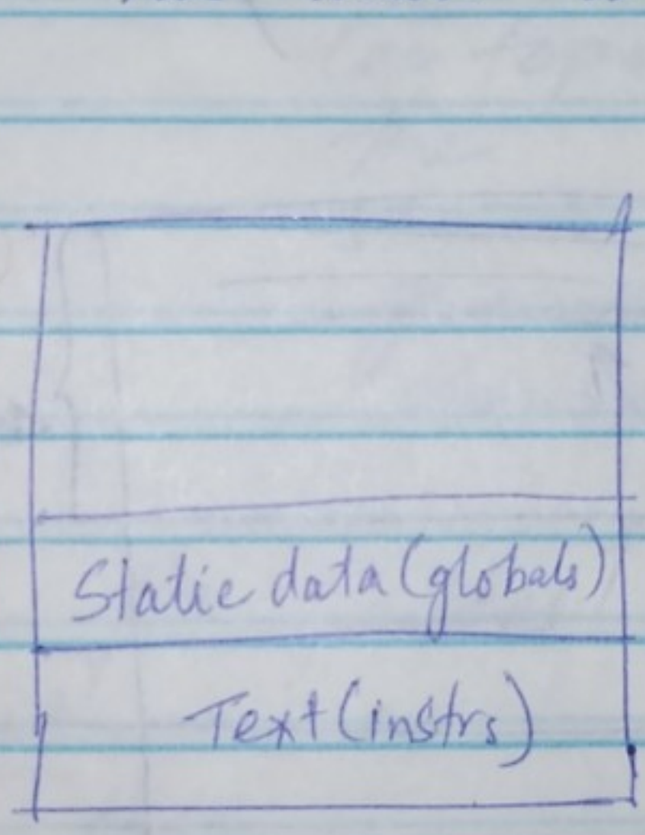
Memory Organisation

FALL 19 CS2.501
Discussion-02
14th Oct 2019

Last time.

Register file in processor, with 32 registers

The program has acess to a larger memory.



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

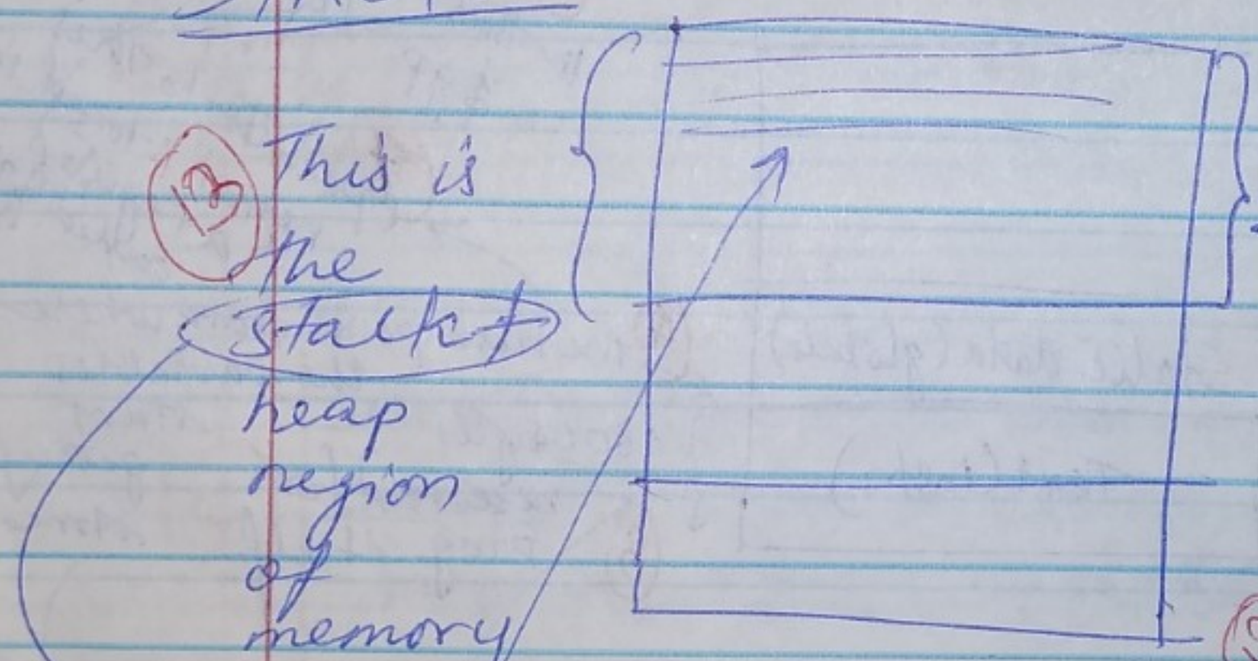
(5) the first set of variables are the global variables. (the very first vars you declare b4 u execute anything)

(9) 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



(11) At compile time compiler does NOT know what gets placed here.

(12) These are determined at runtime.

(13) This is the stack heap region of memory

a
(14) Stores procedures/ fn's variables when u invoke the procedures. — eg.

(15) when you invoke main(), all variables defined inside the fn(). are declared here.

vars only accessible to fn main.

main() may call → find()

those its variables get declared here. (on top of the stack growing downwards)

(16)

(17) → Once the fn finishes they get returns deallocated.

(18) → In this way, the stacks moves up & down

Keeping track of stack is done by the help of 2 pointers. $sp, $fp

start of the procedure

end of the procedure space.

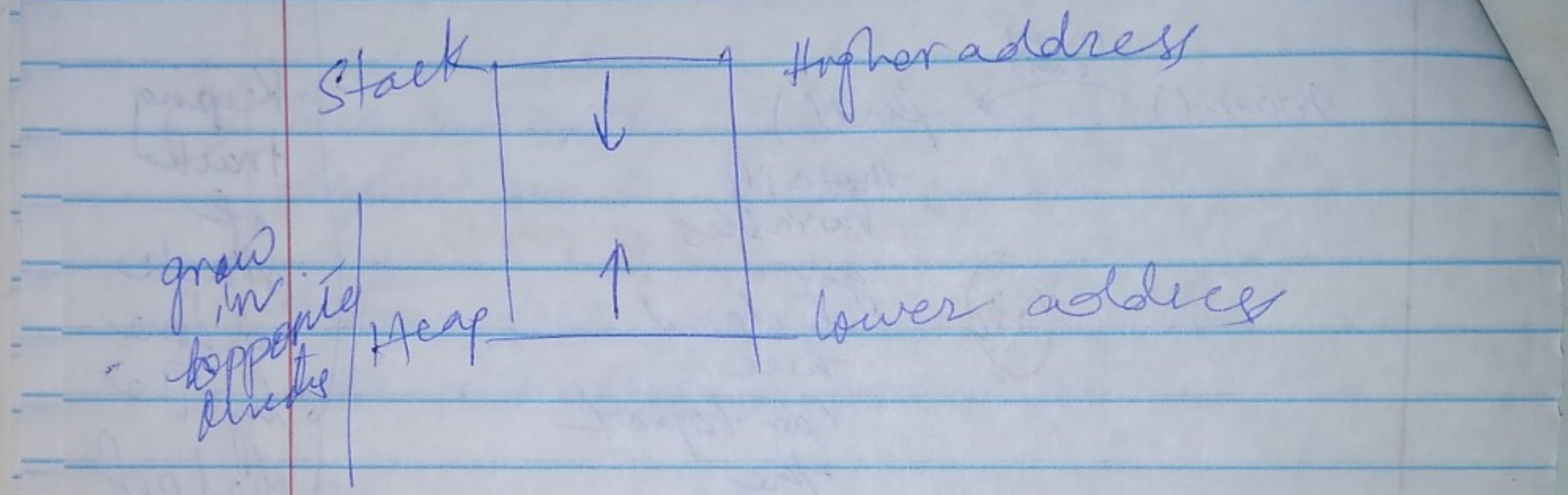
we use one pointer & example

DYNAMIC DATA

(19) Program can dynamically allocate memory — ~~where it defines~~ explicitly — i.e defining how ~~much memory to allocate~~ to allocate.

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

All these vars are stored in heap, & the heap starts where the global variables storage end.



Base address & offsets

E.g C code,

```
a = b + c;
:
int a, b, c, d[10]
main()
{
a = b + c
d[3] = d[2] + a;
}
```

Let's say prog. had 250 instructions, 4 bytes each

code region eats 1000 bytes of space

So global variable starts at address 1000,

So, I do, add 1000 to global pointer

addi $gp, $zero, 1000

need to load b & c,

generate instructions & store new values

lw $s2, 4($gp)

lw $s3, 8($gp)

add $s1, $s2, $s3

sw $s1, $gp

addi $s4, $gp, 12

lw $t0, 8($s4) # get d[2]

add $t0, $t0, $s1

sw $t0, 12($s4)

reg		from $gp
$s1	a	0 from $gp
$s2	b	4 from $gp
$s3	c	8 from $gp
$s4	d	12

Instruction Formats (Have Instr. format slide)

- 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 operations instructions like,

sll
srl

by which where bits are shifted to the left or right.

These bits tell us how much this shift is.

I-type e.g. immediate,
lw, sw

~~Have~~ ~~Fi~~ 5 bits → 1st reg. operand
5 bits → next reg. operand
16 bits → stores the offsets.

beq $s1, $s2, L1

L1: ———

bneq

se slt ⟶ ⊛ not itself a branch instruction
(set on less than) ⊛ sets a certain value based on a comparison of 2 registers.

eg slt $t0, $f1, $f2
set $t0 to 1 if $f1 < $f2

can be used with a bodlow up branch instruction which checks result of slt instruction & proceeds accordingly

2nd kind: Unconditional branch:
jr ⟶ jump to address in a register.

beq $s1, $s2, L1

L1: ———

bneq

se slt ⟶ ⊛ not itself a branch instruction
(set on less than) ⊛ sets a certain value based on a comparison of 2 registers.

eg slt $t0, $f1, $f2
set $t0 to 1 if $f1 < $f2

can be used with a bodlow up branch instruction which checks result of slt instruction & proceeds accordingly

2nd kind: Unconditional branch:
jr ⟶ jump to address in a register.

Task [code]

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

[Assembly]

Say $s1 & $s2
have i & j

[explain the code]

```
while(save[i]
        ==k)
    i+=1;
```

Say, i, k, save(base) are in $s3, $s5, $s6

Ans:

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

Loop:

P.T.O

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

(1) Jump to the top of the loop

(2) get the

(3)
save[0] → is 1 in $s6
save[1] → " " $s6+4 value in
save[2] → " " $s6+8 save[i]

save[i] → " " $s6 + 4×i

(5)
2^2

(4)
shift i by 2 to the left