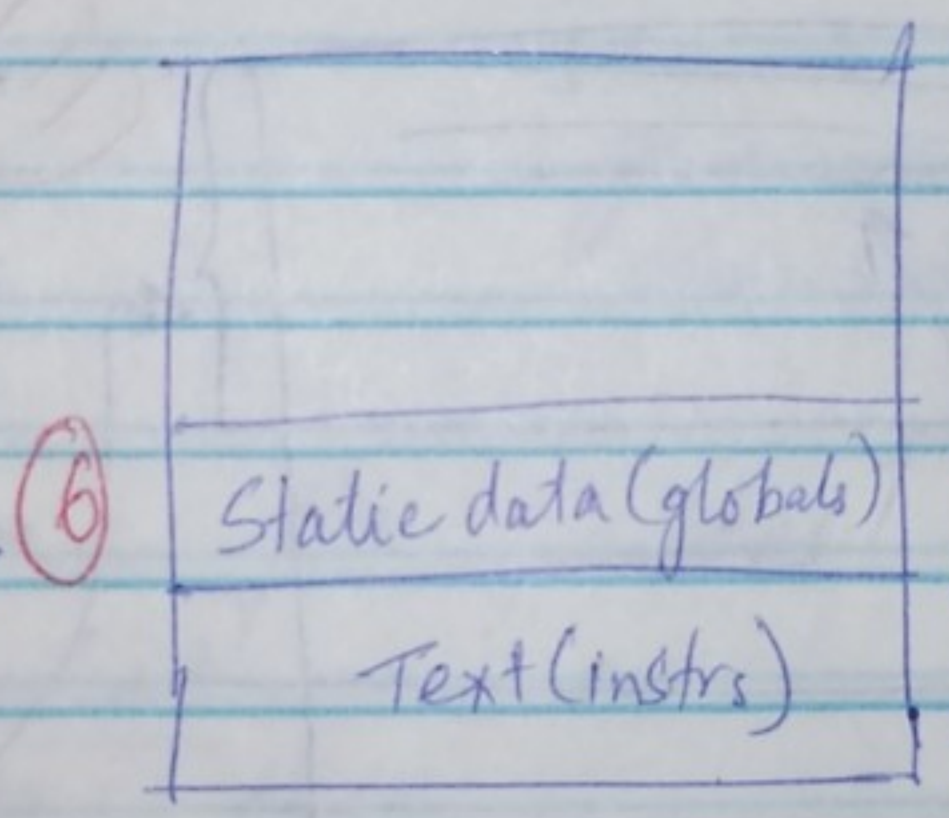
FALL 19 CS250P
Discussion-02
14th Oct 2019

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.

2 Each instr. takes 4 bytes

3 1600 bytes reserved for prog. itself

4 the base point from where the variables start getting stored.

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

7 (pointer to this region stored in a register called $gp)

8 In this example $gp = 1600

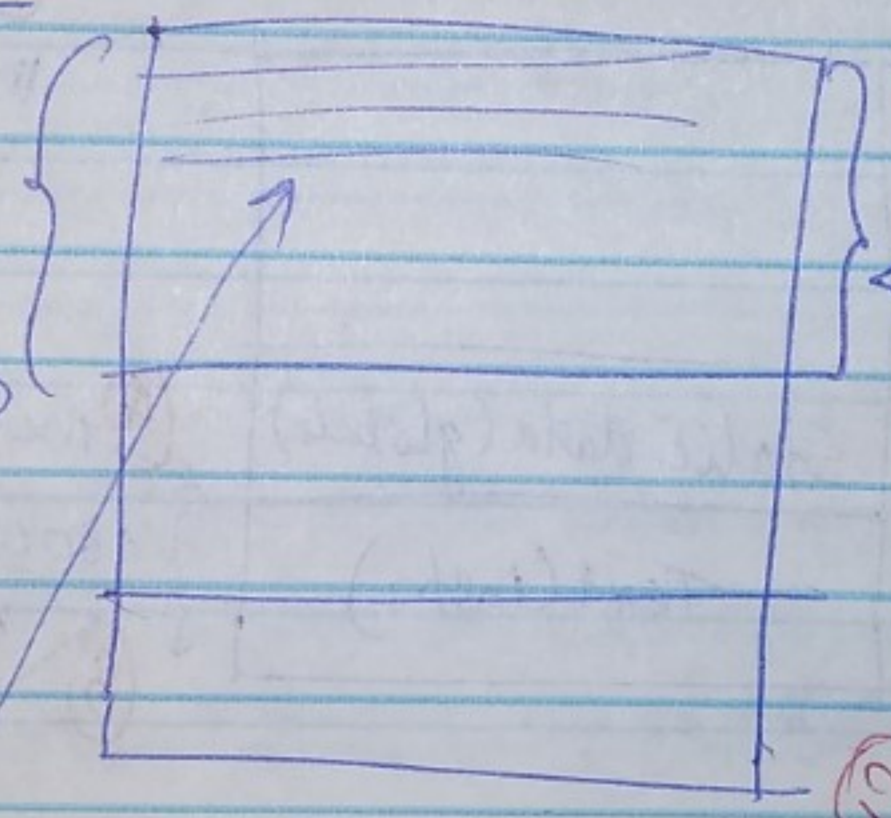
9 The compiler knows all these global vars (the variables that are declared at the start).

(9) These variables get allocated beforehand, in the global region.
(before runtime, during execution)

(10) ~~A~~ From that point on, variables are allocated as per need.

STACK

(13) This is the stack/ heap region of memory



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

(12) These are determined at runtime.

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

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

vars only accessible to fn main.

main() $\xrightarrow{\text{may call}}$ find()

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

17 >Once the f^n finishes / returns they get deallocated.

18 >In this way, the Stacks mones 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.

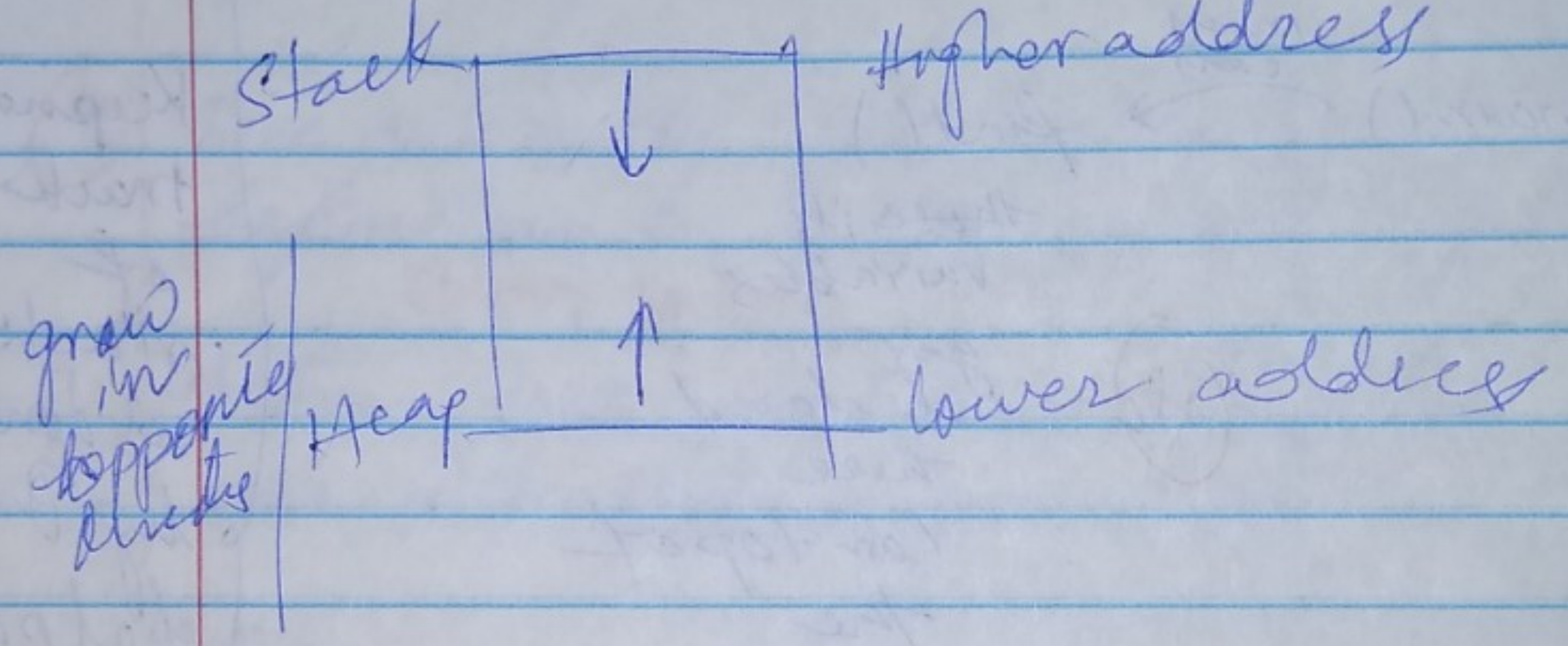
use one f^n 2 ~~example~~

DYNAMIC DATA

19 Program can dynamically allocate memory ~~where & define~~ explicitly — i.e defining how ~~much memory to allocate~~ to allocate.

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

~~A~~ 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,

~~$s0~~ $s9

$

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 (e.g. adding a byte, 2 bytes, or a word)

⊗ (the 5 bits shift is used for ~~operation~~ 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.

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

can be used with a followup 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.

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

can be used with a followup branch instructi which checks result of slt instructin, & proceeds accordingly

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

Task [Code]	[Assembly]

if (i==j)
  f = g+h;
else
  f = g-h;
	Say $s1 & $s2
	have i & j ✓
	
	[explain in slide]

while(save[i] == k)
  i+=1;
	Say, i, k, save(base) are
	into $s3, $s5, $s6
	
	Ans:

A Labels ~~are~~ can also be an instructi (an empty instruction) assembler/ compiler will read it as a ja 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 value in save[i]

(3)

save[0] → 1st in $s6

save[1] → " " $s6+4

save[2] → " " $s6+8

save[i] → " $s6 + 4×i (5)

2^2 → shift i by 2 to the left (4)