CS 316:
Procedure Calls/Pipelining

Kavita Bala

Fall 2007

Computer Science

Cornell University

Announcements

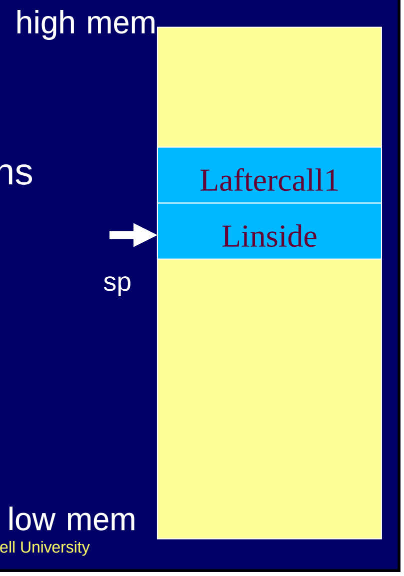
- PA 3 IS out today
  - – Lectures on it this Fri and next Tue/Thu
  - – Due on the Friday after Fall break

Procedures

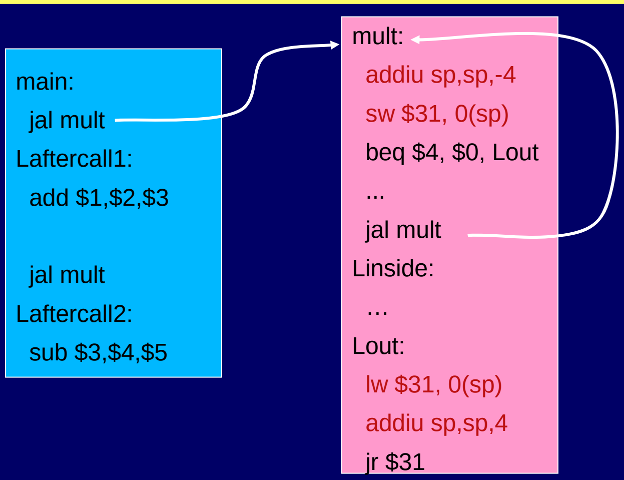
- Enable code to be reused by allowing code snippets to be invoked
- Will need a way to
  - call the routine
  - pass arguments to it
    - fixed length
    - variable length
    - Recursive calls
  - return value to caller
  - manage registers

Call Stacks

- A call stack contains activation records (aka stack frames)
- Each activation record contains
  - the return address for that invocation
  - the local variables for that procedure



Take 3: JAL/JR with Activation Records



```
main:
  jal mult
Laftercall1:
  add $1,$2,$3

  jal mult
Laftercall2:
  sub $3,$4,$5
```

```
mult:
  addiu sp,sp,-4
  sw $31, 0(sp)
  beq $4, $0, Lout
  ...
  jal mult
Linside:
  …
Lout:
  lw $31, 0(sp)
  addiu sp,sp,4
  jr $31
```

- Stack used to save and restore contents of $31

Simple Argument Passing

```
main:
  li a0, 6
  li a1, 7
  jal min
  // result in v0
```

- First four arguments are passed in registers
  - Specifically, $4, $5, $6 and $7, aka a0, a1, a2, a3
- The returned result is passed back in a register
  - Specifically, $2, aka v0

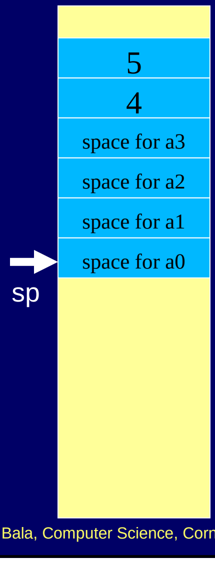
Variable Length Arguments

- Best to use an (initially confusing but ultimately simpler) approach:
  - Pass the first four arguments in registers, as usual
  - Pass the rest on the stack
  - Reserve space on the stack for all arguments, including the first four

- Simplifies functions that use variable-length arguments
  - Store a0-a3 on the slots allocated on the stack, refer to all arguments through the stack

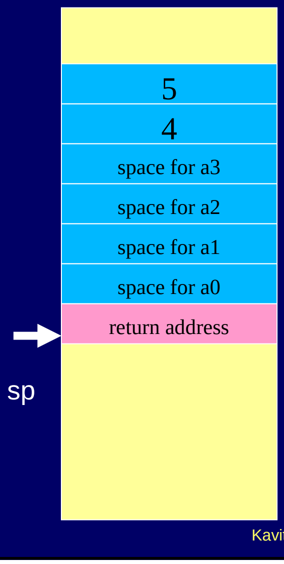
Register Layout on Stack

```
main:
  li a0, 0
  li a1, 1
  li a2, 2
  li a3, 3
  addiu sp,sp,-24
  li $8, 4
  sw $8, 16(sp)
  li $8, 5
  sw $8, 20(sp)
  jal subf
  // result in v0
```



- First four arguments are in registers
- The rest are on the stack
- There is room on the stack for the first four arguments, just in case

Frame Layout on Stack

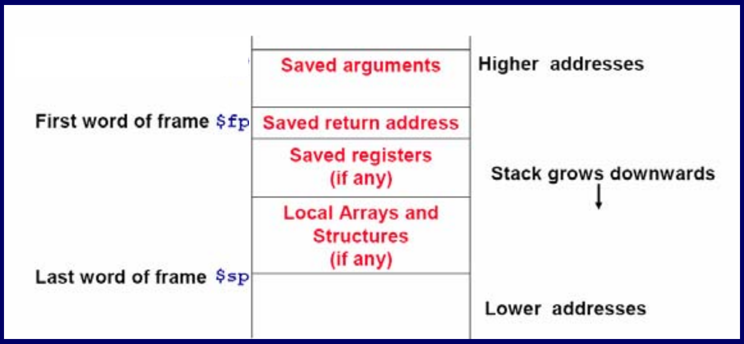


```
blue() {
   pink(0,1,2,3,4,5);
}
pink() {
    ...
}
```

Frame Pointer

- It is sometimes cumbersome to keep track of location of data on the stack
  - The offsets change as new values are pushed onto and popped off of the stack
- Keep a pointer to the top of the stack frame
  - Simplifies the task of referring to items on the stack
- A frame pointer, $30, aka fp
  - Value of sp upon procedure entry
  - Can be used to restore sp on exit

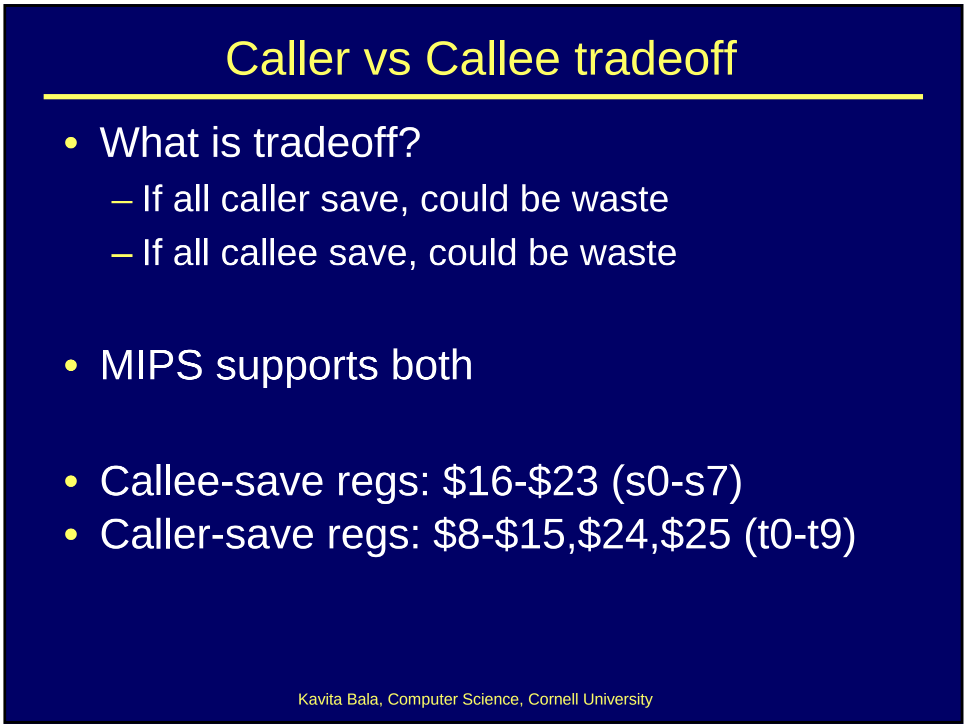
Frame Pointer



Register Usage

- Callee-save
  - Save it if you modify it
  - Assumes caller needs it
  - Save the previous contents of the register on procedure entry, restore just before procedure return
  - E.g. $31 (if you are a non-leaf… what is that?)

- Caller-save
  - Save it if you need it after the call
  - Assume callee can clobber any one of the registers
  - Save contents of the register before proc call
  - Restore after the call

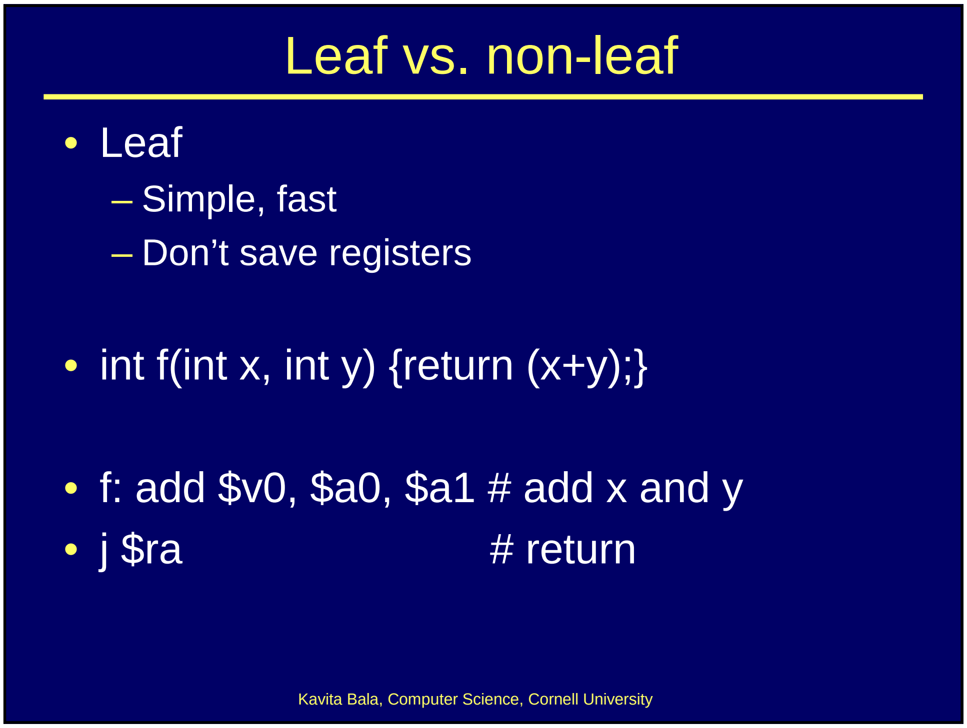


Caller vs Callee tradeoff

- What is tradeoff?
  - If all caller save, could be waste
  - If all callee save, could be waste

- MIPS supports both

- Callee-save regs: $16-$23 (s0-s7)
- Caller-save regs: $8-$15,$24,$25 (t0-t9)



Leaf vs. non-leaf

- Leaf
  - Simple, fast
  - Don’t save registers

- int f(int x, int y) {return (x+y);}

- f: add $v0, $a0, $a1 # add x and y
- j $ra # return

Callee-Save

```
mult:
  addiu sp,sp,-12
  sw $31,8(sp)
  sw $17, 4(sp)
  sw $16, 0(sp)
              …
  [use $17 and $16]
     …
  lw $31,8(sp)
  lw $17, 4(sp)
  lw $16, 0(sp)
  addiu sp,sp,12
```

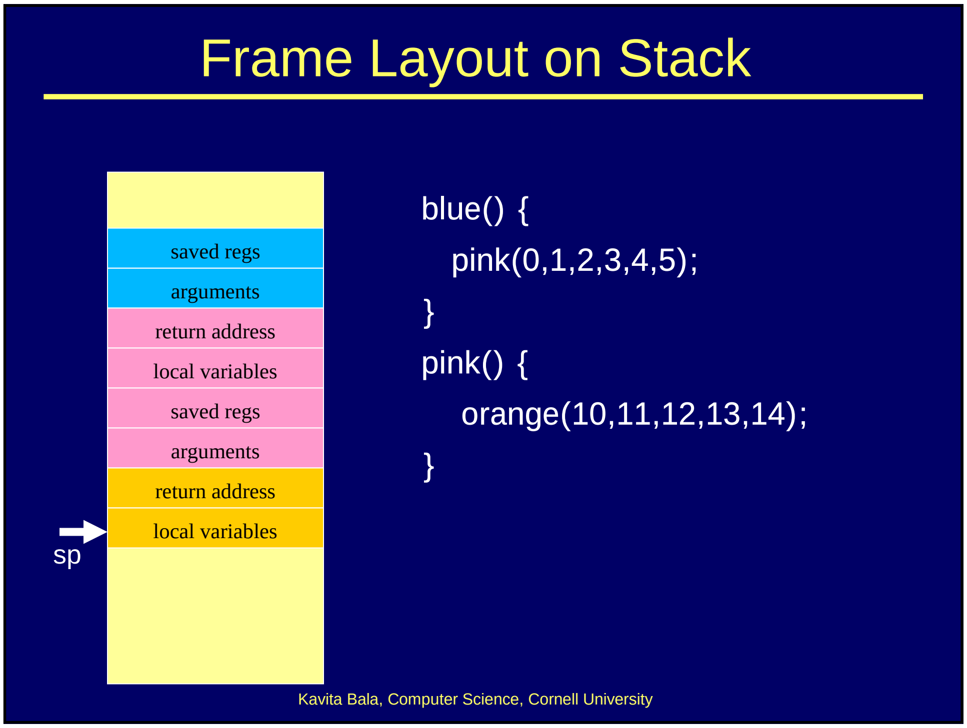
- Assume caller is using the registers
- Save on entry, restore on exit

- Pays off if caller is actually using the registers, else the save and restore are wasted

Caller-Save

```
main:
  …
  [use $9 & $8]
  …
  addiu sp,sp,-8
  sw $9, 4(sp)
  sw $8, 0(sp)
  jal mult
  lw $9, 4(sp)
  lw $8, 0(sp)
  addiu sp,sp,8
  …
  [use $9 & $8]
```

- Assume registers are free for the taking
- But other subroutines will do the same
  - must protect values that will be used later
  - save and restore them before and after subroutine invocations
- Pays off if a routine makes few calls to other routines with values that need to be preserved

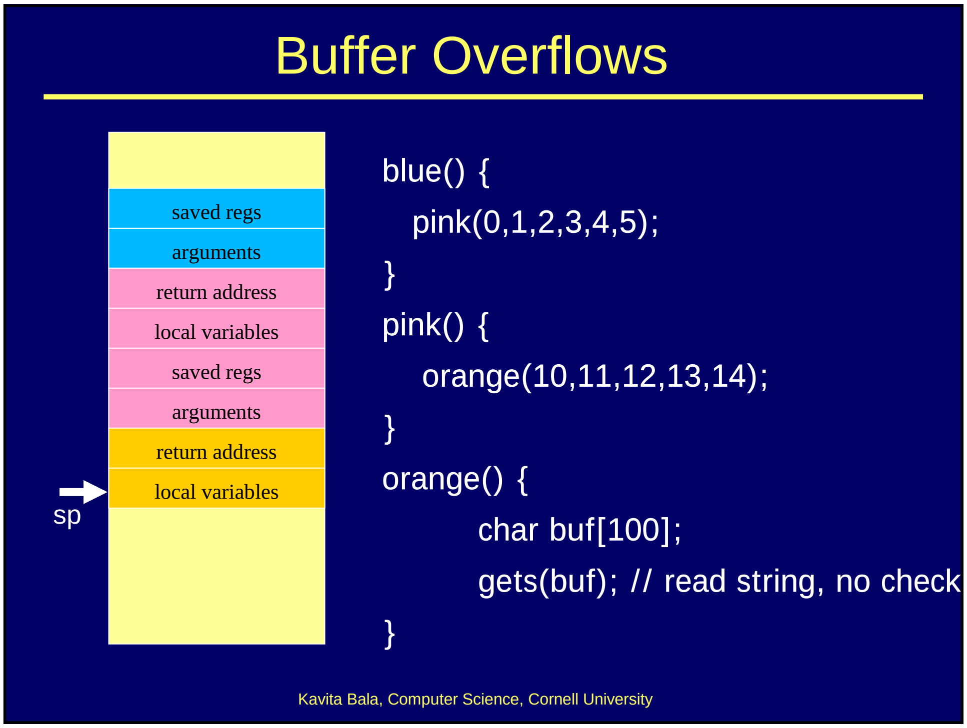


Frame Layout on Stack

saved regs
arguments
return address
local variables
saved regs
arguments
return address
local variables

```
blue() {
   pink(0,1,2,3,4,5);
}
pink() {
    orange(10,11,12,13,14);
}
```

Kavita Bala, Computer Science, Cornell University

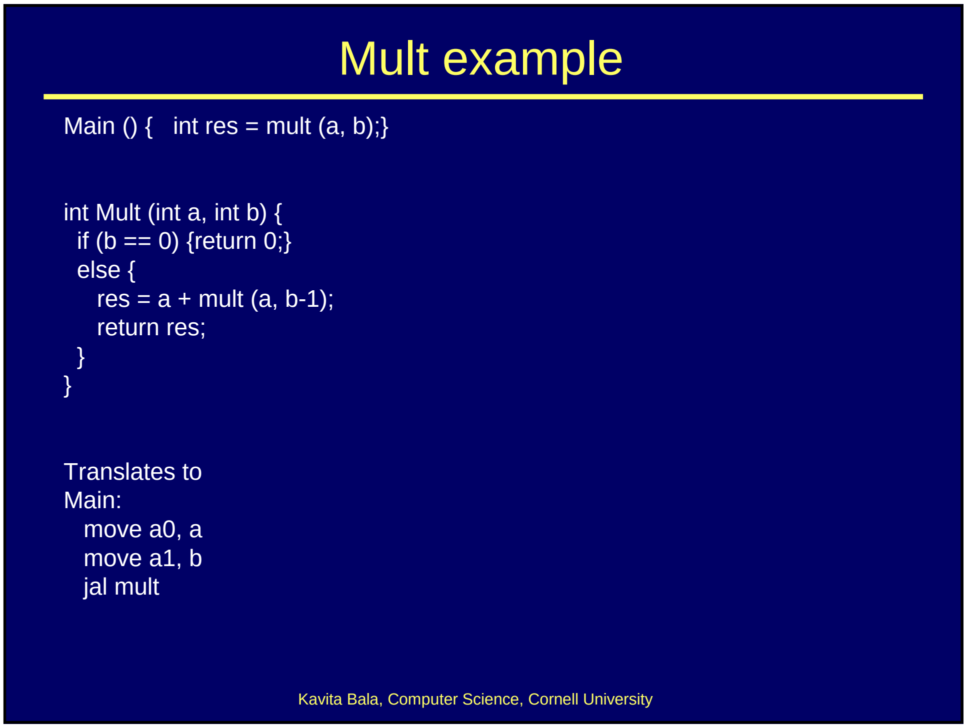


Buffer Overflows

saved regs
arguments
return address
local variables
saved regs
arguments
return address
local variables

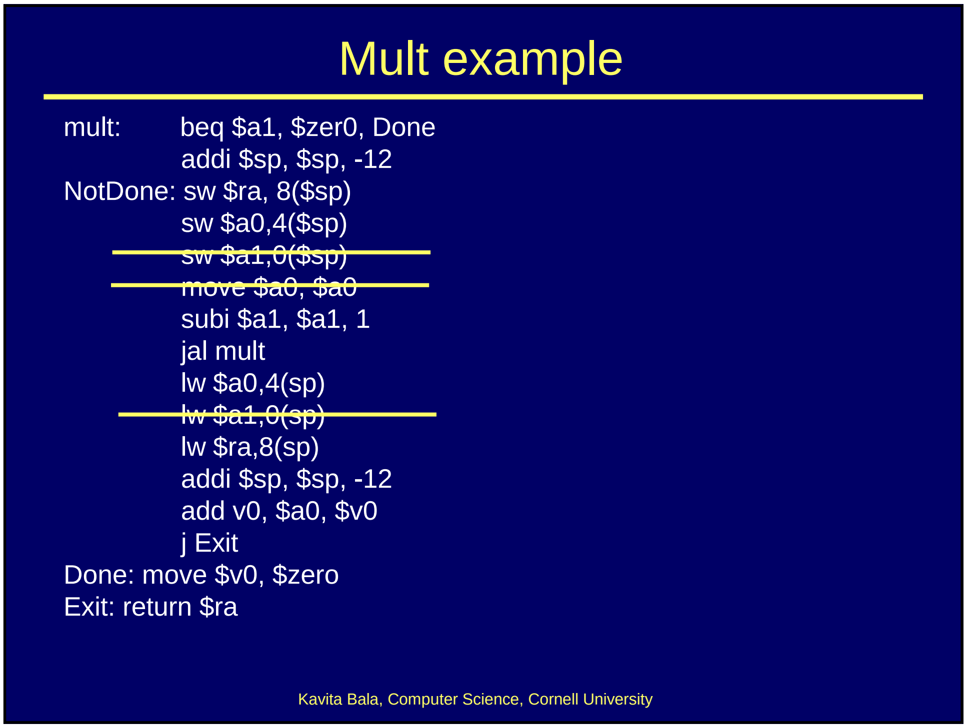
```
blue() {
   pink(0,1,2,3,4,5);
}
pink() {
    orange(10,11,12,13,14);
}
orange() {
         char buf[100];
         gets(buf); // read string, no check
}
```

Kavita Bala, Computer Science, Cornell University



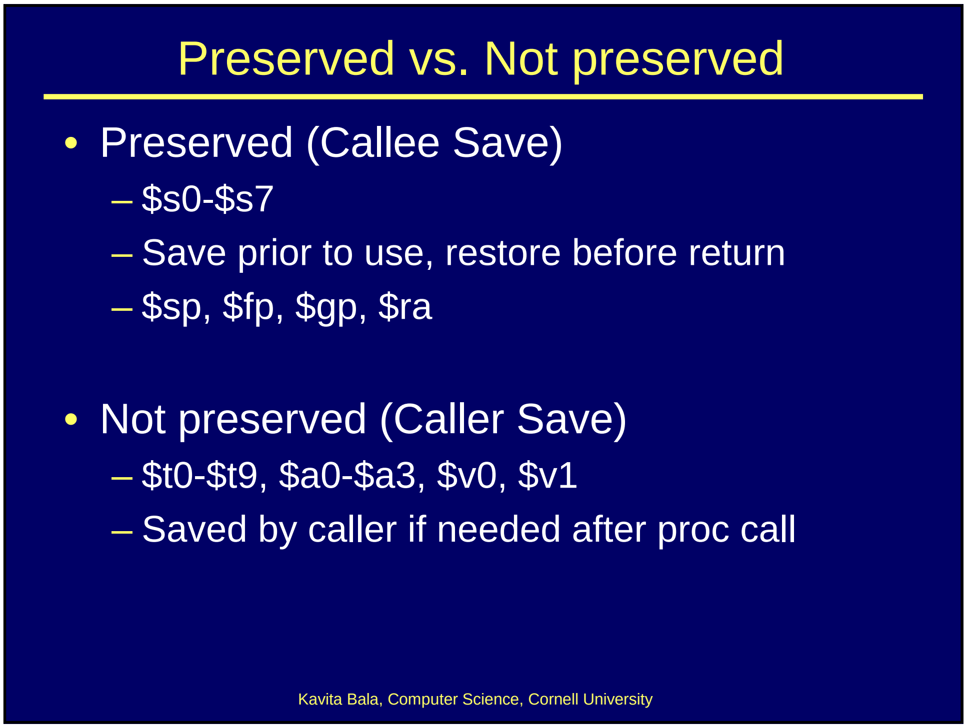
Mult example

```
Main () {   int res = mult (a, b);}


int Mult (int a, int b) {
  if (b == 0) {return 0;}
  else {
     res = a + mult (a, b-1);
     return res;
  }
}


Translates to
Main:
   move a0, a
   move a1, b
   jal mult
```



Mult example

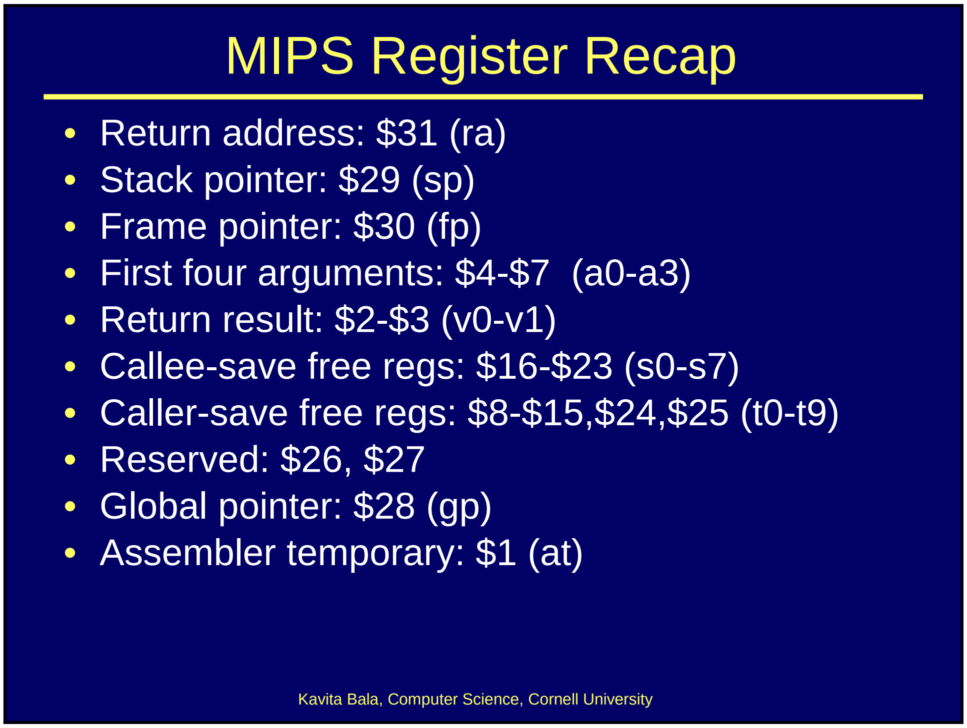
```
mult:      beq $a1, $zer0, Done
           addi $sp, $sp, -12
NotDone: sw $ra, 8($sp)
           sw $a0,4($sp)
           ~~sw $a1,0($sp)~~
           ~~move $a0, $a0~~
           subi $a1, $a1, 1
           jal mult
           lw $a0,4(sp)
           ~~lw $a1,0(sp)~~
           lw $ra,8(sp)
           addi $sp, $sp, -12
           add v0, $a0, $v0
           j Exit
Done: move $v0, $zero
Exit: return $ra
```



Preserved vs. Not preserved

- Preserved (Callee Save)
  - $s0-$s7
  - Save prior to use, restore before return
  - $sp, $fp, $gp, $ra

- Not preserved (Caller Save)
  - $t0-$t9, $a0-$a3, $v0, $v1
  - Saved by caller if needed after proc call

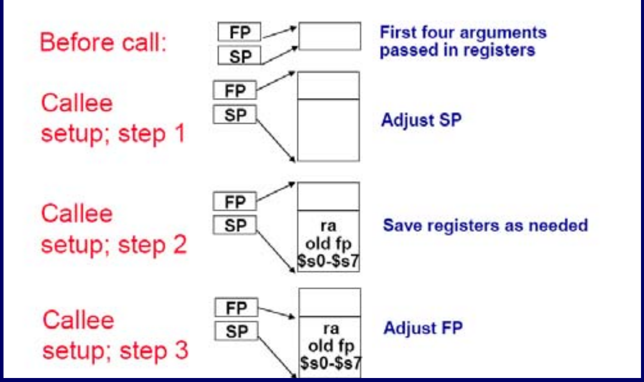


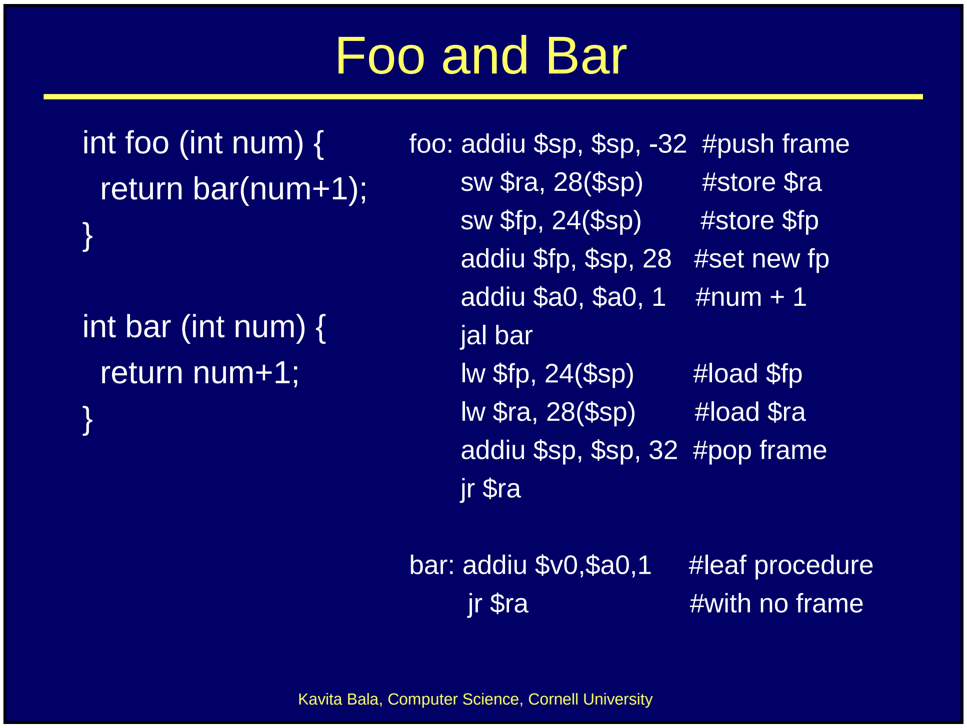
MIPS Register Recap

- Return address: $31 (ra)
- Stack pointer: $29 (sp)
- Frame pointer: $30 (fp)
- First four arguments: $4-$7 (a0-a3)
- Return result: $2-$3 (v0-v1)
- Callee-save free regs: $16-$23 (s0-s7)
- Caller-save free regs: $8-$15,$24,$25 (t0-t9)
- Reserved: $26, $27
- Global pointer: $28 (gp)
- Assembler temporary: $1 (at)

What happens on a call?

- Caller
  - Save caller-saved registers \$a0-\$a3, \$t0-\$t9
  - Load arguments in \$a0-\$a3, rest passed on stack
  - Execute jal
- Callee Setup
  - Allocate memory for new frame (\$sp = \$sp-frame)
  - Save callee-saved registers \$s0-\$s7, \$fp, \$ra
  - Set frame pointer (\$fp = \$sp + frame size – 4)
- Callee Return
  - Place return value in \$v0 and \$v1
  - Restore any callee-saved registers
  - Pop stack (\$sp = \$sp + frame size)
  - Return by jr \$ra

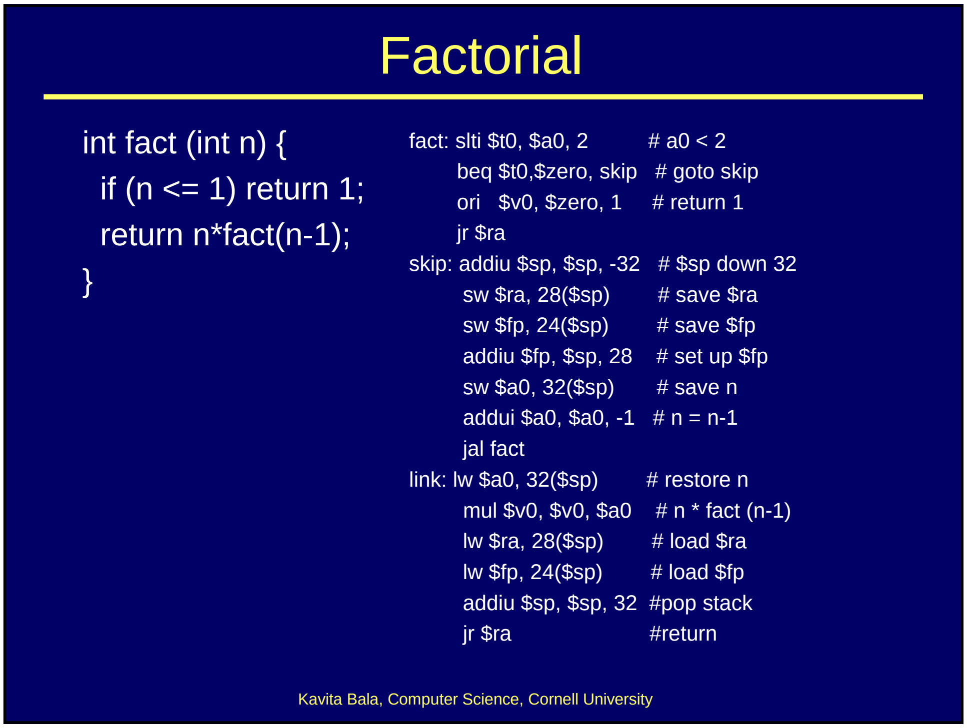




Foo and Bar

```
int foo (int num) {
  return bar(num+1);
}

int bar (int num) {
  return num+1;
}
```

```
foo: addiu $sp, $sp, -32  #push frame
     sw $ra, 28($sp)      #store $ra
     sw $fp, 24($sp)      #store $fp
     addiu $fp, $sp, 28   #set new fp
     addiu $a0, $a0, 1    #num + 1
     jal bar
     lw $fp, 24($sp)      #load $fp
     lw $ra, 28($sp)      #load $ra
     addiu $sp, $sp, 32   #pop frame
     jr $ra

bar: addiu $v0,$a0,1      #leaf procedure
      jr $ra              #with no frame
```



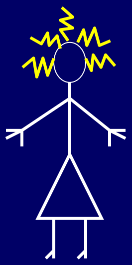
Factorial

```
int fact (int n) {
  if (n <= 1) return 1;
  return n*fact(n-1);
}
```

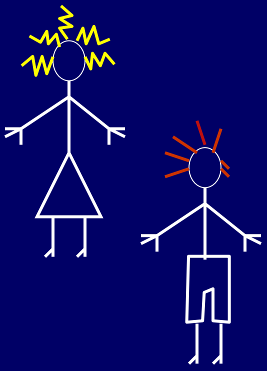
```
fact: slti $t0, $a0, 2          # a0 < 2
      beq $t0,$zero, skip       # goto skip
      ori   $v0, $zero, 1       # return 1
      jr $ra
skip: addiu $sp, $sp, -32       # $sp down 32
       sw $ra, 28($sp)          # save $ra
       sw $fp, 24($sp)          # save $fp
       addiu $fp, $sp, 28       # set up $fp
       sw $a0, 32($sp)          # save n
       addui $a0, $a0, -1       # n = n-1
       jal fact
link: lw $a0, 32($sp)           # restore n
       mul $v0, $v0, $a0        # n * fact (n-1)
       lw $ra, 28($sp)          # load $ra
       lw $fp, 24($sp)          # load $fp
       addiu $sp, $sp, 32       #pop stack
       jr $ra                   #return
```

Pipelined Architectures

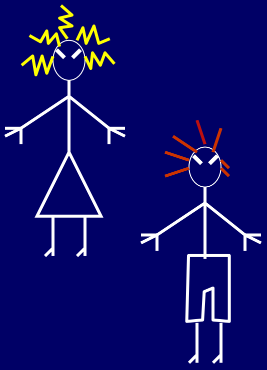
- Alice



- Alice
- Bob



- Alice
- Bob



- They don’t like each other!

The Laundry

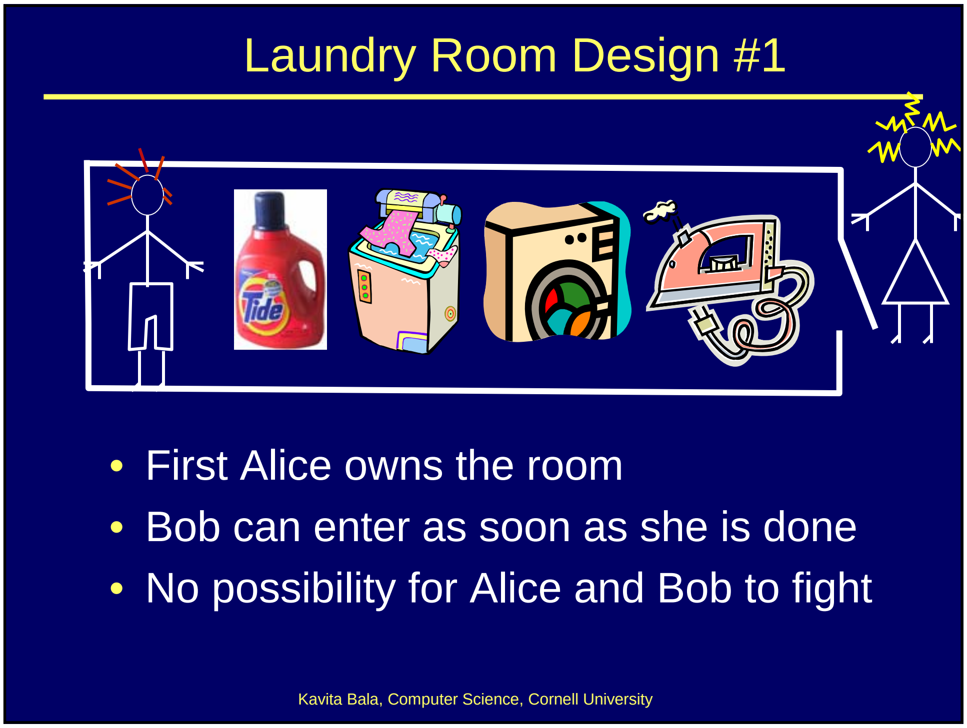
- Four sequential tasks

Laundry Room Design #1

- A large room with a one entry-door and one exit-door

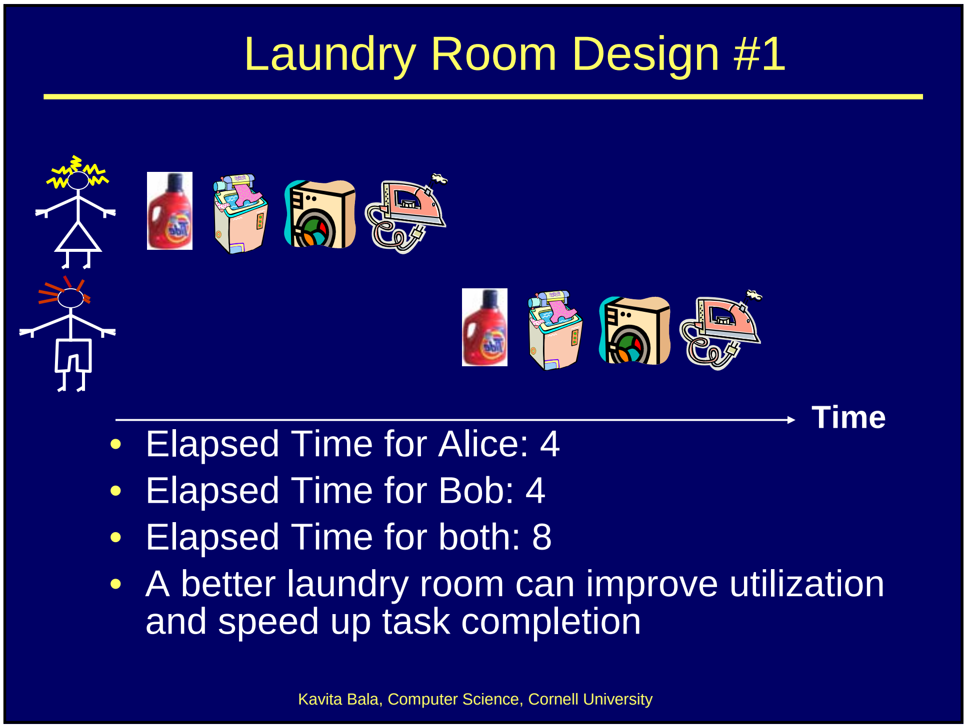
Laundry Room Design #1

- First Alice owns the room



Laundry Room Design #1

- First Alice owns the room
- Bob can enter as soon as she is done
- No possibility for Alice and Bob to fight

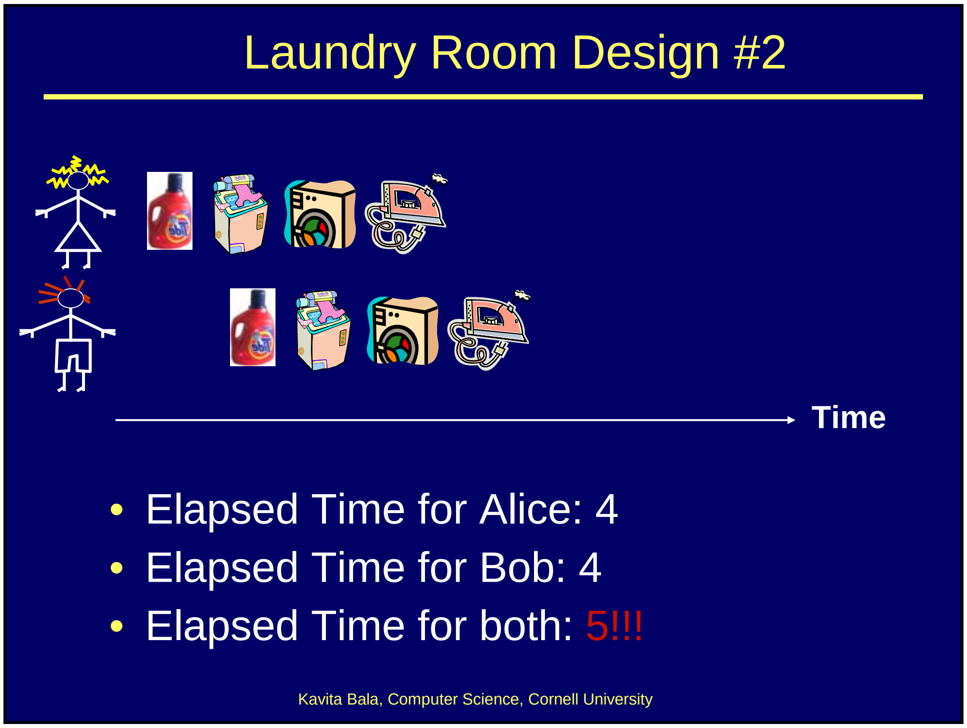


Laundry Room Design #1

Time

- Elapsed Time for Alice: 4
- Elapsed Time for Bob: 4
- Elapsed Time for both: 8
- A better laundry room can improve utilization and speed up task completion

Kavita Bala, Computer Science, Cornell University



Laundry Room Design #2

Time

- Elapsed Time for Alice: 4
- Elapsed Time for Bob: 4
- Elapsed Time for both: 5!!!

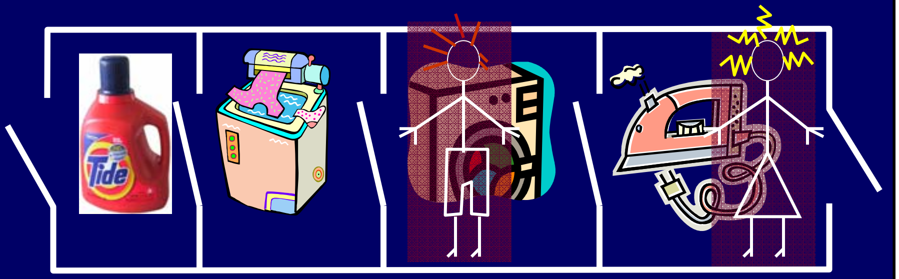
Kavita Bala, Computer Science, Cornell University

Laundry Room Design #2

- The room is partitioned into stages
- One person owns a stage at a time, the room can hold up to four people simultaneously

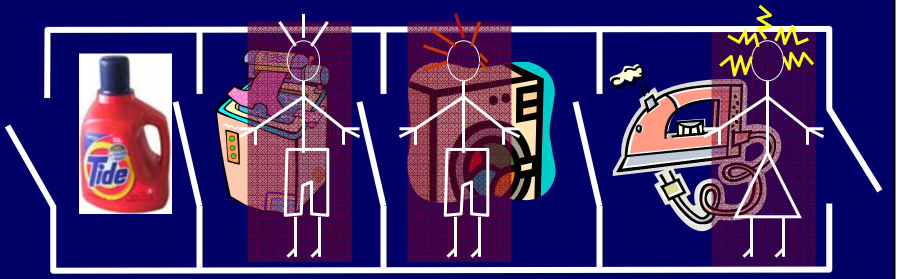
Laundry Room Design #2

Laundry Room Design #2

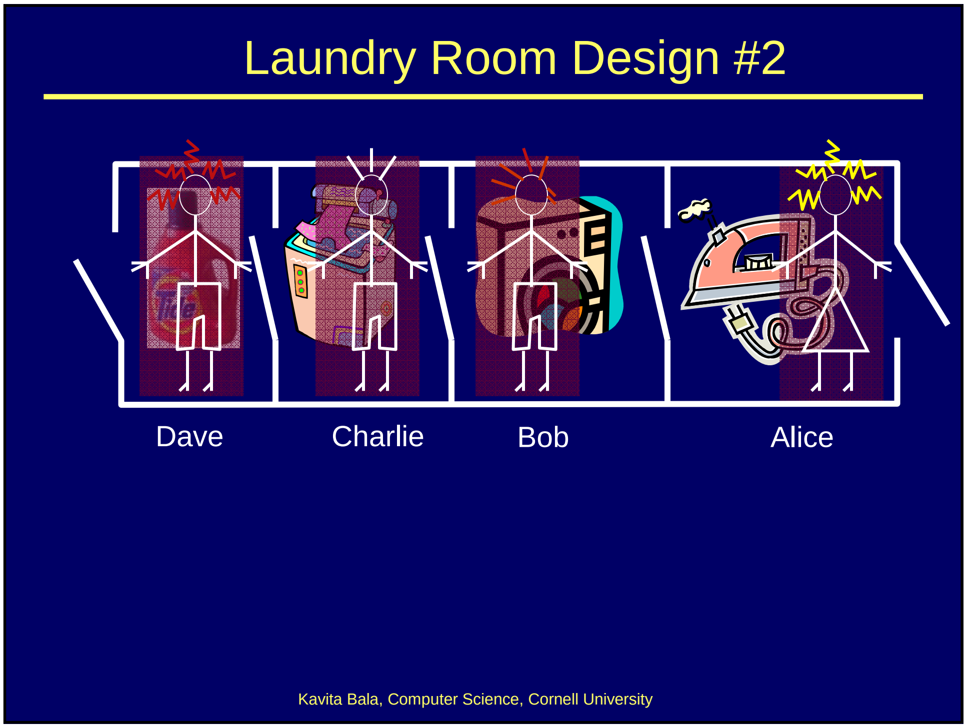


Kavita Bala, Computer Science, Cornell University

Laundry Room Design #2

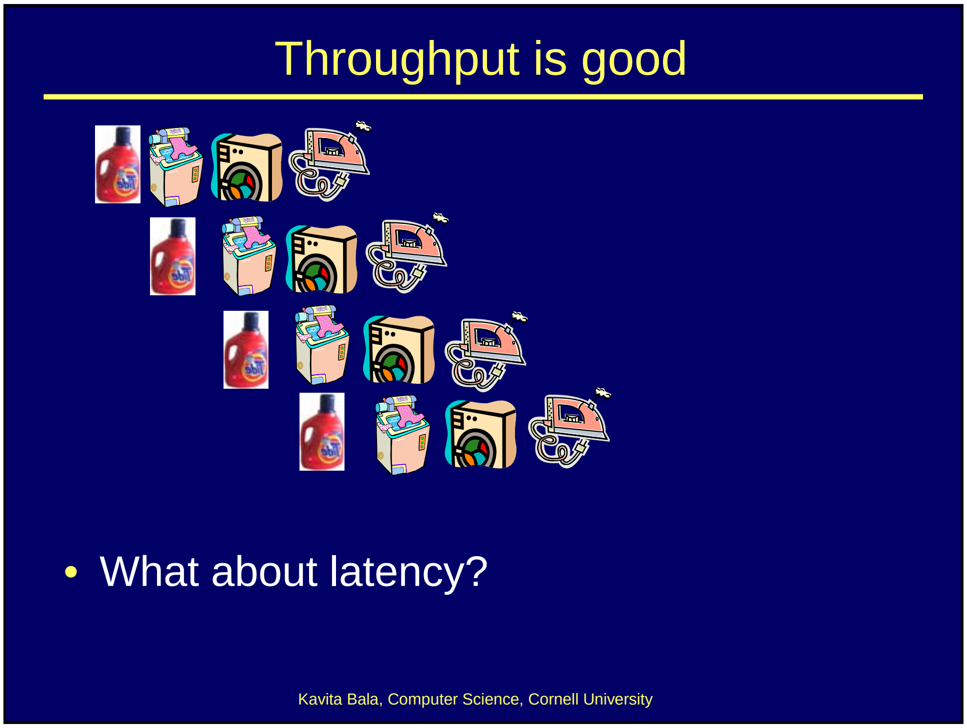


Kavita Bala, Computer Science, Cornell University



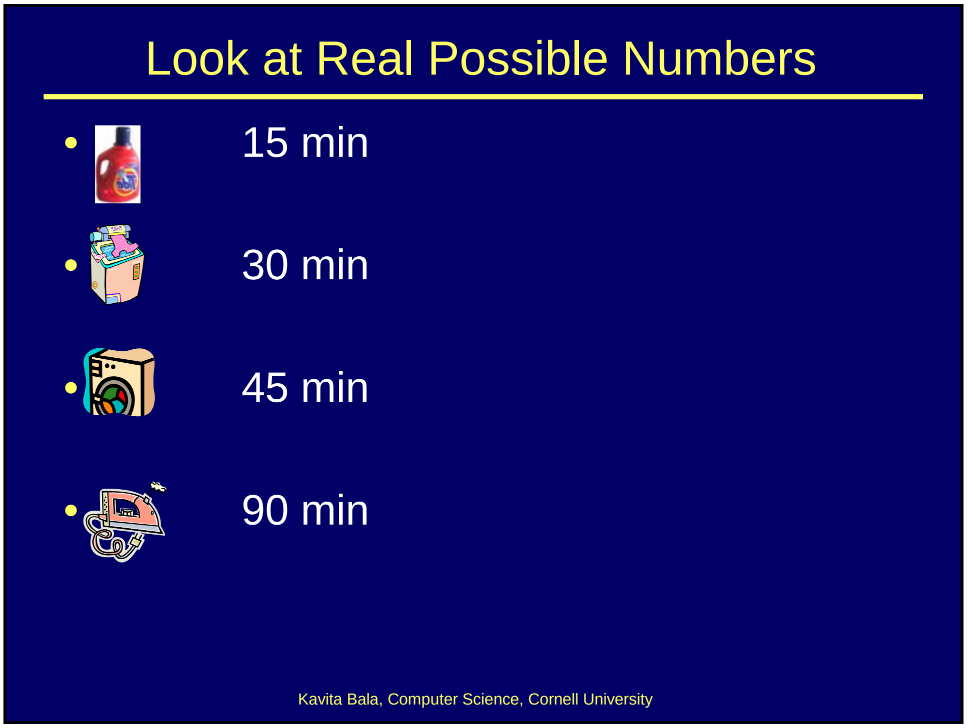
Laundry Room Design #2

Dave Charlie Bob Alice



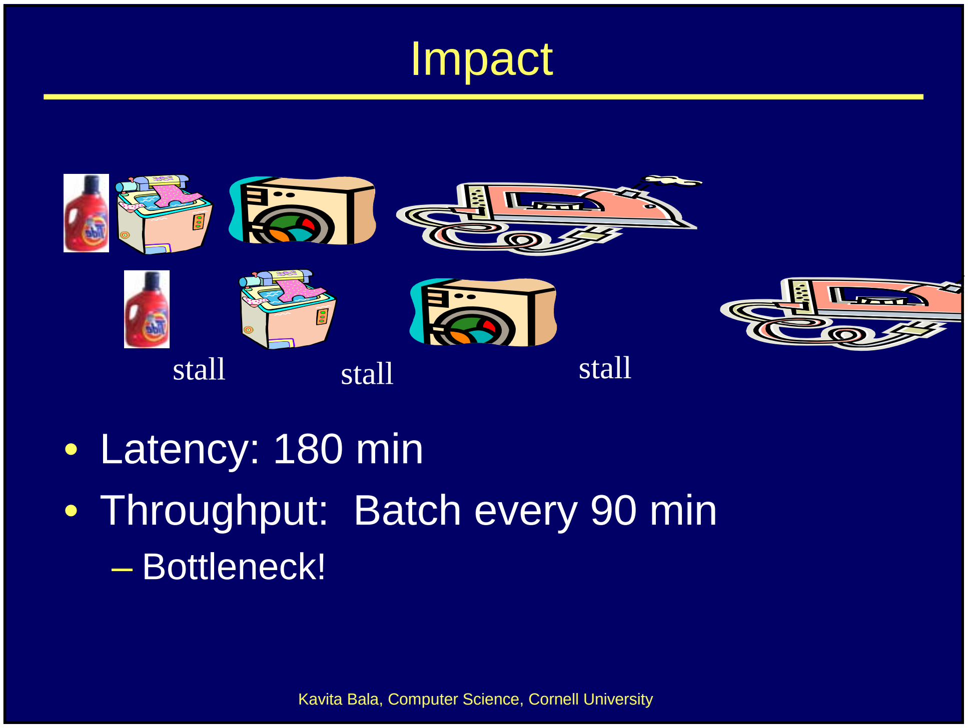
Throughput is good

- What about latency?



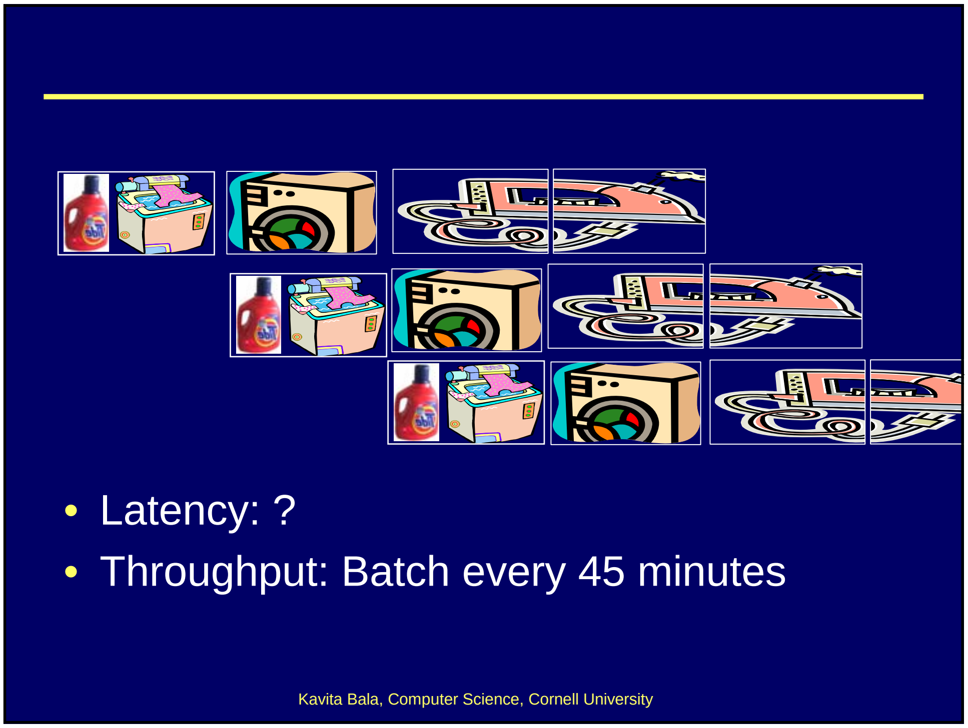
Look at Real Possible Numbers

- 15 min
- 30 min
- 45 min
- 90 min

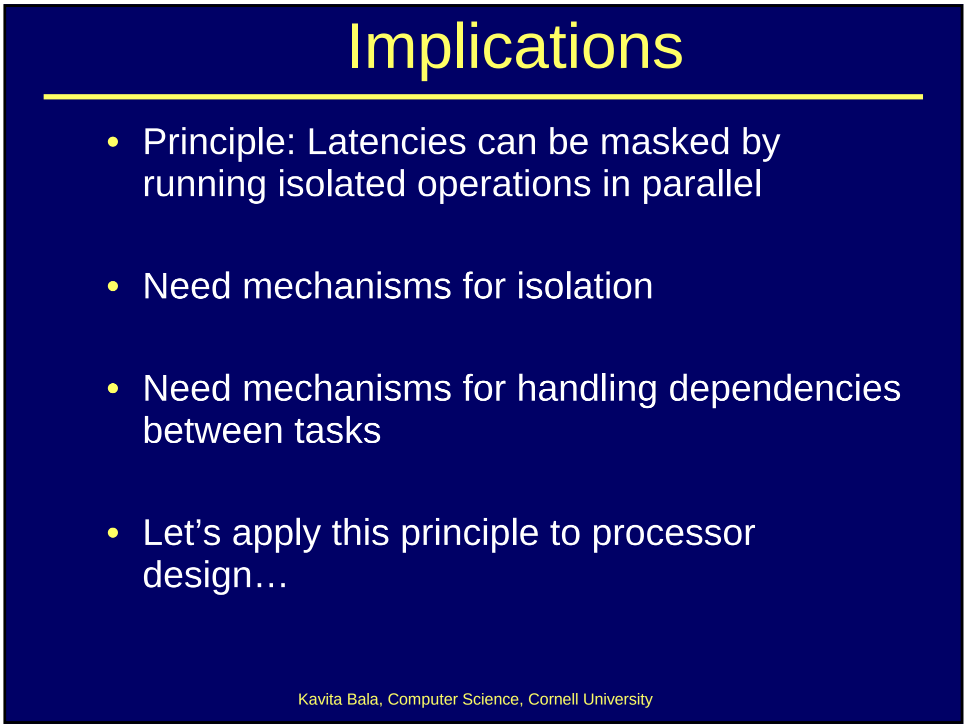


Impact

- Latency: 180 min
- Throughput: Batch every 90 min
  - Bottleneck!



- Latency: ?
- Throughput: Batch every 45 minutes



Implications

- Principle: Latencies can be masked by running isolated operations in parallel
- Need mechanisms for isolation
- Need mechanisms for handling dependencies between tasks
- Let’s apply this principle to processor design…