9 September 2004

- Questions?
- Programming Continued

So far we've learned:

- MIPS
  - loading words but addressing bytes
  - arithmetic on registers only
- Instruction / Meaning

```
add $s1, $s2, $s3      $s1 = $s2 + $s3
sub $s1, $s2, $s3      $s1 = $s2 – $s3
lw $s1, 100($s2)       $s1 = Memory[$s2+100]
sw $s1, 100($s2)       Memory[$s2+100] = $s1
```


Execution Example

Program Counter: 200

Memory(32 bits):

200	10001101000010010000000000000000	LW $9, 0($8)
204	00000001001001110100100000100000	ADD $9,$9,$7
208	10101101000010010000000000001000	SW $9, 8($8)
212		

Memory(32 bits):

112	8
116	13
120	21
124	34
128	55
132	89

Registers (32 bits):

6	1234
7	23
8	120
9	-314159
10	316

Instruction Register (32 bits):

	op	rs	rt	rd	shft	func
R	6 bits	5 bits	5 bits	5 bits	5 bits	6 bits

	op	rs	rt	offset
I	6 bits	5 bits	5 bits	16 bits

Execution Example: Fetch(200)

Program Counter: 200

Memory:

200	10001101000010010000000000000000	LW $9, 0($8)
204	00000001001001110100100000100000	ADD $9,$9,$7
208	10101101000010010000000000001000	SW $9, 8($8)
212		

Memory:

112	8
116	13
120	21
124	34
128	55
132	89

Registers:

6	1234
7	23
8	120
9	-314159
10	316

Instruction Register:

R						
I	100011	01000	01001	0000000000000000		
	35	8	9	0		

Execution Example: Execute(200)

Program Counter: 204

Memory:

200	10001101000010010000000000000000	LW $9, 0($8)
204	00000001001001110100100000100000	ADD $9,$9,$7
208	10101101000010010000000000001000	SW $9, 8($8)
212		

Memory:

112	8
116	13
120	21
124	34
128	55
132	89

Registers:

6	1234
7	23
8	120
9	21
10	316

Instruction Register:

R						
I	100011	01000	01001	0000000000000000		
	35	8	9	0		

Execution Example: Fetch(204)

Program Counter: 204

Memory:

200	10001101000010010000000000000000	LW $9, 0($8)
204	00000001001001110100100000100000	ADD $9,$9,$7
208	10101101000010010000000000001000	SW $9, 8($8)
212		

Memory:

112	8
116	13
120	21
124	34
128	55
132	89

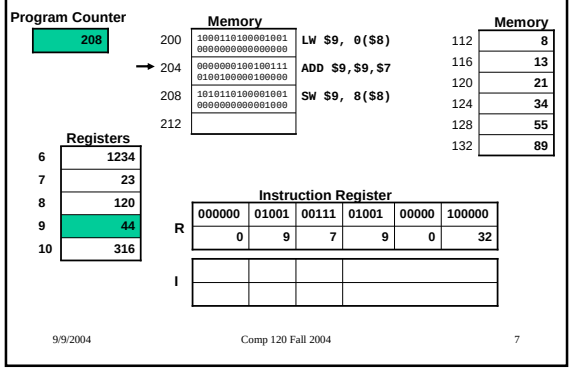
Registers:

6	1234
7	23
8	120
9	21
10	316

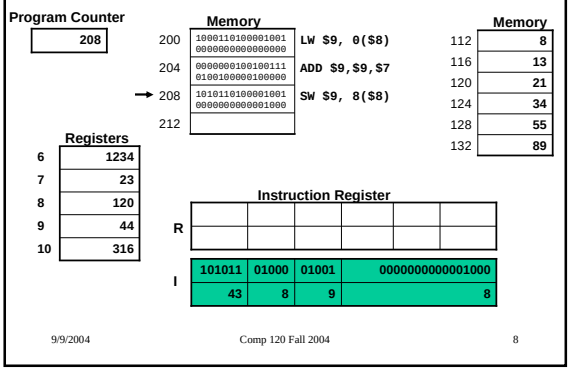
Instruction Register:

R	000000	01001	00111	01001	00000	100000
	0	9	7	9	0	32
I						

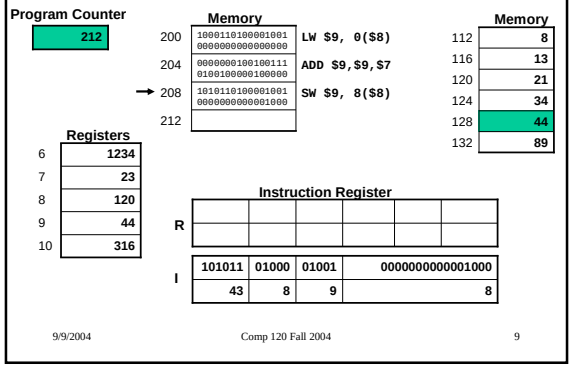
Execution Example: Execute(204)



Execution Example: Fetch(208)



Execution Example: Execute(208)



Control

- Decision making instructions
  - alter the control flow,
  - change the "next" instruction to be executed by changing the PC
- MIPS conditional branch instructions:

```
bne $t0, $t1, Label
beq $t0, $t1, Label
```

- Example: if (i==j) h = i + j;

```
        bne $s0, $s1, Label
        add $s3, $s0, $s1
Label: ....
```


Control

- MIPS unconditional branch instructions:

```
j  label
jal label          # function call
jr $ra             # function return
```

- Example:

```
if (i!=j)          beq $s4, $s5, Lab1
    h=i+j;         add $s3, $s4, $s5
else               j Lab2
    h=i-j;         Lab1:  sub $s3, $s4, $s5
                   Lab2:  ...
```


Control Flow

- We have: beq, bne, what about Branch-if-less-than?
- New instruction:

```
                          if  $s1 < $s2 then
                             $t0 = 1
slt $t0, $s1, $s2         else
                             $t0 = 0
```

- Can use this instruction to build "**`blt $s1, $s2, Label`**"
  — can now build general control structures
- Note that the assembler needs a register to do this,
  — there are policy of use conventions for registers

So far:

- Instruction — Meaning

```
add $s1,$s2,$s3   $s1 = $s2 + $s3
lw $s1,100($s2)   $s1 = Memory[$s2+100]
bne $s4,$s5,L     Next instr. is at Label if $s4 != $s5
j Label           Next instr. is at Label
jal label         Next instr at label, save return addr
jr $ra            Next instr is addr in ra
```

- Formats:

R	op	rs	rt	rd	shamt	funct
I	op	rs	rt	16 bit address		
J	op	26 bit address				

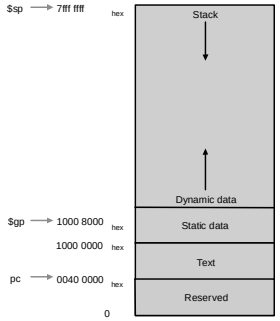
Policy of Use Conventions

Name	Register number	Usage
$zero	0	the constant value 0
$at	1	reserved for the assembler
$v0-$v1	2-3	values for results and expression evaluation
$a0-$a3	4-7	arguments
$t0-$t7	8-15	temporaries
$s0-$s7	16-23	saved
$t8-$t9	24-25	more temporaries
$gp	28	global pointer
$sp	29	stack pointer
$fp	30	frame pointer
$ra	31	return address

Stack Pointer?

- Register $sp is used to keep track of the **stack**
- Nothing special about $sp in the hardware, just a convention
- Stack starts at the TOP of memory and grows DOWN (Why?)
- Allocate space on the stack by decrementing $sp
- Free space on the stack by incrementing $sp
- Used to save return address and temporary variables as well as for *local* variables and arrays

Memory Layout



Making code work!

- This example is available online.
- You should write algorithms in C/JAVA first
  - Then HAND translate…

C to Assembler

```
char S1[] = "This is a string."
int Foo[16];
char S2[100];

void main()
 {
   strcpy(S2, S1);
 }

void strcpy(char* dst, char* src)
 {
   int i = 0;
   while((dst[i] = src[i]) != 0)
     i = i+1;
 }
```

Data Segment

```
#char S1[] = "This is a string."
#int Foo[16];
#char S2[100];

       .data
S1:    .asciiz      "This is a string."
Foo:   .space 16                # some junk
S2:    .space 100               # destination string
```


strcpy

```
#void strcpy(char* dst, char* src) {
#   int i = 0;
#   while((dst[i] = src[i]) != 0)
#     i = i+1;
# }

strcpy:
   move $t0,$zero      # i in $t0 = 0
L1:
   add  $t1,$a1,$t0    # address of src[i] in $t1
   lb   $t2, 0($t1)    # t2 = src[i]
   add  $t1,$a0,$t0    # address of dst[i] in $t1
   sb   $t2, 0($t1)    # dst[i] = $t2
   addi $t0, $t0, 1    # i = i+1
   bne  $t2,$zero,L1   # if dst[i] != 0 repeat

   jr   $ra            # return
```


Text Segment (main)

```
#void main() { strcpy(S2, S1); }

   .text
   .globl main
main:
   addi$sp, $sp, -4 # get space on the stack
   sw  $ra, 0($sp)  # save main's return address
   la  $a0, S2      # address of S2 in the first argument
   la  $a1, S1      # address of S1 in the second argument
   jal strcpy       # call strcpy
   lw  $ra, 0($sp)  # restore main's return address
   addi$sp, $sp, 4  # restore the stack pointer
   jr  $ra          # exit main
```


How to write assembly programs

- Develop the algorithm in a higher-level language (C/Java/Whatever)
- Translate statement by statement to assembly
- Use “Code Patterns” to guide the translation
- Only a few idioms to master
- Then it is just a mechanical process

Code Pattern for IF

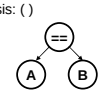
```
if(COND_EXPR) { STMTS1 } else { STMTS2 }

   CONDCODE(COND_EXPR, Lfalse1, Ltrue2)
Ltrue2:
   CODE(STMTS1)
   j   Lnext3
Lfalse1:
   CODE(STMTS2)
Lnext3:
```


Code Pattern for Conditional Expr

Comparisons: == != < > <= >=
Conjunctions: && ||
Parenthesis: ( )

A == B




```
get left in reg A
get right in reg B
bne $A,$B, Lfalse
```

A < B


```
get left in reg A
get right in reg B
slt $t0,A,B
beq t0,$zero,Lfalse
```

CP for && and ||

L && R

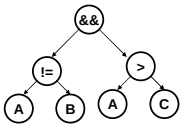

```
        CONDCODE(L, Lfalse, Lnew1)
Lnew1:
        CONDCODE(R, Lfalse, Ltrue)
```

L || R


```
        CONDCODE(L, Lnew1, Ltrue)
Lnew1:
        CONDCODE(R, Lfalse, Ltrue)
```


Example Conditional

if(A != B && A > C) { ST } else { SE }




```
        A=s1, B=s2, C=s3
        beq    $s1,$s2,Lfalse23
Lnew24:
        slt $t0, $s3, $s1
        beq $t0, $zero, Lfalse23
Ltrue25:
        CODE(ST)
        j      Lnext27
Lfalse23:
        CODE(SE)
Lnext27:
```


While

while(COND_EXPR) { STMTS; }

```
Lwhile44:
  CONDCODE(COND_EXPR, Lfalse45, Ltrue46)
Ltrue46:
  CODE(STMTS)
  j   Lwhile44
Lfalse45:
```


FOR

```
for(INIT; COND_EXPR; UPDATE) { STMTS }

  CODE(INIT)
Lfor17:
  CONDCODE(COND_EXPR, Lnext18, Ltrue19)
Ltrue19:
  CODE(STMTS)
  CODE(UPDATE)
  j   Lfor17
Lnext18:
```


Functions

int foo(int a, int b, int c, int d) { STMTS; return EXPR; }

we know a=$a0, b=$a1, c=$a2, d=$a3
assign other simple variables to registers as possible

```
foo:
  save any of $ra or $s0-s7 that are overwritten
  CODE(STMTS)
  CODE(EXPR) leave result in $v0
  restore $ra, and $s0-s7 if we saved them earlier
  jr  $ra
```


cfind

```
int cfind(char str[], char c) {
  for(int i=0; str[i] != 0; i++)
      if(str[i] == c) return i;
  return -1;
}

a0 == address of str
a1 == value of character c
v0 == i

no need to save anything
```

Expand function template

```
cfind:
  # no need to save anything
  CODE('for ...')
  CODE('return -1')
  jr  $ra
```


Expand for

```
cfind:
  CODE('i=0')
Lfor1:
  CONDCODE('s[i] != 0, Lnext2, Ltrue3)
Ltrue3:
  CODE('if ...')
  CODE('i++')
  j   Lfor1
Lnext2:
  CODE('return -1')
  jr  $ra
```


Expand for INIT

```
cfind:
  move      $v0, $zero
Lfor1:
  CONDCODE('s[i] != 0', Lnext2, Ltrue3)
Ltrue3:
  CODE('if ...')
  CODE('i++')
  j   Lfor1
Lnext2:
  CODE('return -1')
  jr  $ra
```


Expand for COND_EXPR

```
cfind:
  # no need to save anything
  move      $v0, $zero
Lfor1:
  add $t0, $a0, $v0           # address of s[i]
  lb  $t1, 0($t0)             # t1 = s[i]
  beq $t1, $zero, Lnext2
Ltrue3:
  CODE('if ...')
  CODE('i++')
  j   Lfor1
Lnext2:
  CODE('return -1')
  jr  $ra
```


Expand if

```
cfind:
  move$v0, $zero
Lfor1:
  add $t0, $a0, $v0       # address of s[i]
  lb  $t1, 0($t0)               # t1 = s[i]
  beq $t1, $zero, Lnext2
Ltrue3:
  CONDCODE('s[i] == c', Lfalse4, Ltrue5)
Ltrue5:
  CODE('return i');
Lfalse4:
  CODE('i++')
  j   Lfor1
Lnext2:
  CODE('return -1')
  jr  $ra
```


Expand if COND_EXPR

```
cfind:
  move$v0, $zero
Lfor1:
  add $t0, $a0, $v0       # address of s[i]
  lb  $t1, 0($t0)               # t1 = s[i]
  beq $t1, $zero, Lnext2
Ltrue3:
  bne $t1, $a1, Lfalse4
Ltrue5:
  CODE('return i');
Lfalse4:
  CODE('i++')
  j   Lfor1
Lnext2:
  CODE('return -1')
  jr  $ra
```

Expand return

```
cfind:
   move$v0, $zero
Lfor1:
   add $t0, $a0, $v0       # address of s[i]
   lb  $t1, 0($t0)               # t1 = s[i]
   beq $t1, $zero, Lnext2
Ltrue3:
   bne $t1, $a1, Lfalse4
Ltrue5:
   jr  $ra   # return i already in v0
Lfalse4:
   CODE('i++')
   j   Lfor1
Lnext2:
   CODE('return -1')
   jr  $ra
```


Expand i++

```
cfind:
   move$v0, $zero
Lfor1:
   add $t0, $a0, $v0       # address of s[i]
   lb  $t1, 0($t0)               # t1 = s[i]
   beq $t1, $zero, Lnext2
Ltrue3:
   bne $t1, $a1, Lfalse4
Ltrue5:
   jr  $ra    # return i already in v0
Lfalse4:
   addi      $v0, $v0, 1       # i = i+1;
   j   Lfor1
Lnext2:
   CODE('return -1')
   jr  $ra
```


Expand return -1

```
cfind:
   move$v0, $zero
Lfor1:
   add $t0, $a0, $v0       # address of s[i]
   lb  $t1, 0($t0)               # t1 = s[i]
   beq $t1, $zero, Lnext2
Ltrue3:
   bne $t1, $a1, Lfalse4
Ltrue5:
   jr  $ra    # return i already in v0
Lfalse4:
   addi$v0, $v0, 1         # i = i+1;
   j   Lfor1
Lnext2:
   subi      $v0, $zero, 1     # return value = -1
   jr  $ra
```


Remove unused labels

```
cfind:
   move      $v0, $zero        # i = 0
Lfor1:
   add $t0, $a0, $v0           # address of s[i]
   lb  $t1, 0($t0)             # t1 = s[i]
   beq $t1, $zero, Lnext2      # if s[i] == 0
   bne $t1, $a1, Lfalse4       # if s[i] == c
   jr  $ra   # return i already in v0
Lfalse4:
   addi      $v0, $v0, 1       # i = i+1;
   j   Lfor1
Lnext2:
   subu      $v0, $zero, 1     # return -1
   jr  $ra
```