Mips Code Examples

- ***Peter Rounce***
  P.Rounce@cs.ucl.ac.uk

Some C Examples

Assignment : *int j = 10 ; // space must be allocated to variable j*

Possibility 1: j is stored in a register, i.e. register $2

then the MIPS assembler for this is :-

addi $2, $0, 10 : $2 <- $0 + sign-extend[10]

Possibility 2: j is stored in memory, i.e. memory 0x12345678

then the MIPS assembler for this is :-

lui	$1, 0x1234	: $1 ← 0x12340000	Get address in $1
ori	$1, $1, 0x5678	: $1 ← 0x12345678	
addi	$8, $0, 10	: $8 ← $0 + sign-extend[10]	Get 10 in $8
sw	$8, 0($1)	: Mem[$1 + 0] ← $8	Store 10 → 0x12345678

Program to calculate Absolute value of difference between 2 input numbers: |A - B| (demonstrates if)

Program reads A from 4 bytes of memory starting at address 12345670 (hex).

Program reads B from 4 bytes of memory starting at address 12345674(hex).

Program writes |A-B| to 4 bytes of memory starting at address 12345678(hex).

```
Assembler                          # Comment
lui   $10, 0x1234
ori   $10, $10, 0x5670             # load of A address into register $10
lw    $4, 0($10)                   # read A from memory into register $4
lw    $5, 4($10)                   # read B from memory into register $5
sub   $12, $5, $4                  # subtract A from B => B-A into register $12
bgez  $12,+4                       # branch if B-A is positive to 'sw' instruction
sub   $12, $4, $5                  # subtract B from A => A-B into register $12
sw    $12, 8($10)                  # store register $12 value, |A-B|, into memory
```

Given the binary for an instruction e.g.:

10101101111010001000000000000000

What code would you write to get the r_s register number into a register on its own, and in the low bits of this register?

code

in $4: 10101101111010001000000000000000

in $5: 00000011111000000000000000000000 ***and***

R_s bits

in $6: 00000001111000000000000000000000

Shift right logical

in $6: 00000000000000000000000000001111

What is wanted.

Code: assume code in $4

// get *masking* value in $5

```
lui     $5, 0x03e0

and     $6, $5, $4
```

// masked value in $6

// so shift $6 right

```
srl     $6, $6, 21
```

Change r_s register in instruction, e.g. to register 21 in code:

10101101111010001000000000000000

code

in \$4: 10101101111010001000000000000000

in \$5: 11111100000111111111111111111111

and

in \$6: 10101100000010001000000000000000

in \$5: 00000000000000000000000000010101

in \$5: 00000010101000000000000000000000

or

in \$6: 10101110101010001000000000000000

What is wanted.

Code: assume code in \$4

```
// get masking value in $5
lui     $5, 0xfc1f
ori     $5, $5, 0xffff

and     $6, $5, $4

// new value into $5
addiu   $5, $0, 0x15
sll     $5, $5, 21

or      $6, $6, $5
```

Shift Instructions:

Shift left logical: **sll rd, rt, shift-amount**

rd ← rt << shift-amount : 0s placed on right

Example: Let $4 == 2, then

sll $5, $4, 3

shifts the contents of $4 left 3 places (2<<3)→ 16 which is stored in $5.

	0	0	0	0	0	0	0	0	0	0	0	0	1	0
1:shift left 1	0	0	0	0	0	0	0	0	0	0	0	1	0	0
2:shift left 1	0	0	0	0	0	0	0	0	0	0	1	0	0	0
3:shift left 1	0	0	0	0	0	0	0	0	0	1	0	0	0	0

shift left logical variable: **sllv rd, rt, rs**

: rs holds shift- amount for shifting rt with result into rd

: rd ← rt << rs

---

shift right logical: reverse of shift left logical

srl rd, rt, shift-amount : 0s placed on left

e.g. $5 = 16 then srl $6, $5, 3 : $6 ← 16 >> 3

$6 == 1 after instruction

---

shift right logical variable: **srlv sll rd, rt, rs** **as sllv but right**

Shift right arithmetic: another reverse of shift left logical

shift right arithmetic: **sra rd, rt, shift-amount**

shift right arithmetic variable: **srav rd, rt, rs**

arithmetic shifts duplicate the sign bit : 1s are placed on right for -ve values

1111110000 (>> 2) →1111111100 0011110000 (>> 2) →0000111100

Branches - a ***Reminder!!!!!***

Instructions are always 4 bytes long in Mips.

Instructions are always stored at addresses that are an integer multiple of 4: 0, 4, 8, … 0x2C, 0x30, …. 0x12345678, 0x1234567C…..

pc always points at an instruction,
i.e. pc always holds a multiple of 4

Branches always change pc by a multiple of 4

Branch offset is number of instructions to branch,
not number of addresses!

Branch target address calculation:- pc + offset *4

Conditional Branch Instructions – using labels

calculating offsets is difficult – use a label instead!

you write this

assembler calculates this

Branch Equal

beq rs, rt, Label

: if rs == rt pc <- pc + (address of label – pc)

: if rs == rt pc <- pc + offset*4

Branch Not-Equal

bne reg1, reg2, Label

: if rs != rt pc <- pc + (address of label – pc)

: if rs != rt pc <- pc + offset*4

Assembler Program calculates difference between address of instruction following the branch and the address of Label (label address – pc)/4 and stores this values in offset field of instruction

6 Bits **op**	5 Bits **rs**	5 Bits **rt**	**16-bit offset**

Other Branches

These branches test the contents of a single register against 0.

branch on greater than or equal zero:

bgez register, label : if (register >= 0) pc ← address of label
: if (register >= 0) pc ←pc + offset*4

branch on greater than zero:

bgtz register, label : if (register > 0) pc ← address of label
: if (register > 0) pc ←pc + offset*4

branch on less than or equal zero:

blez register, label : if (register <= 0) pc ← address of label
: if (register <= 0) pc ←pc + offset*4

branch on less than zero:

bltz register, label : if (register < 0) pc ← address of label
: if (register > 0) pc ←pc + offset*4

Note: branches can go –32768 instructions back & 32767 forward
memory address space in Mips is 1G instructions!!

What about comparing 2 registers for < and >=?

Use a Set instruction followed by a conditional branch.

Comparison Instructions

R-Format versions: compare 2 register and put result into 3rd register

Set less than (signed): **slt rd, rs, rt** : if rs<rt set rd=1 else set rd=0

Set less than unsigned: **sltu rd, rs, rt** : if rs<rt set rd=1 else set rd=0

I-Format versions: compare register and constant, put result into 2nd register

Set less than immediate (signed): **slti rd, rs, imm** : if rs<imm set rd=1 else set rd=0

Set less than unsigned immediate: **sltui rd, rs, imm** : if rs<imm set rd=1 else set rd=0

The immediate value, (imm), is 16-bits and is sign-extended to 32 bits before comparison.

Use *beq* or *bne* against *reg $0* to test result register rd after *set*.

MIPS ‘for loop’ *example*

Setting the elements of an array to zero

Data declarations:-

```
unsigned i ;
int array[10] ;
```

N.B. C creates the space for both these automatically
no *new* required.

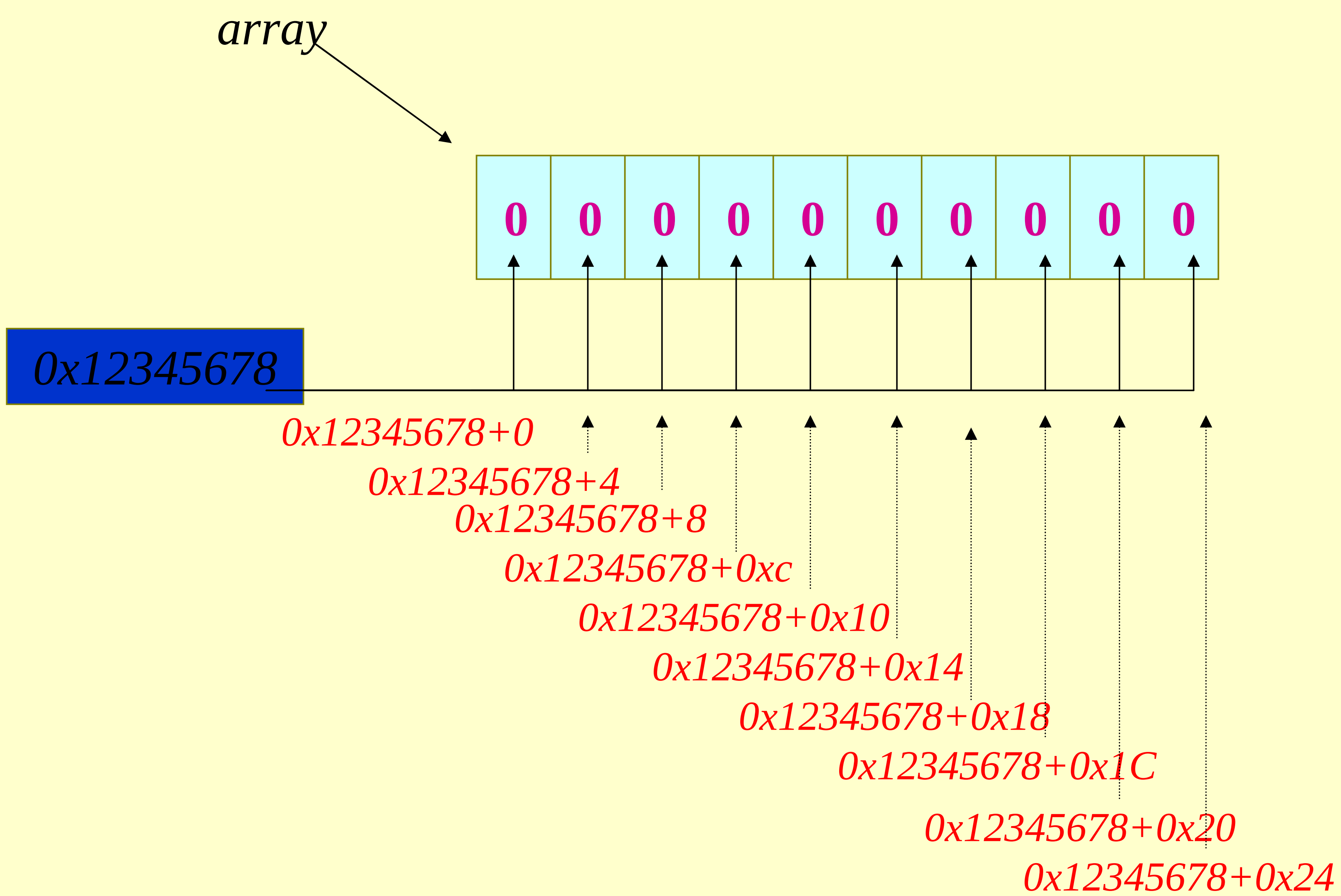
```
for (i=0; i<10; i++) {
        array[i] = 0 ;
}
```

MIPS 'for loop' example

Let the variable i be stored in register \$4
Let 'int array' start at address 12345678_{16}
Each integer occupies 4 addresses
Use \$8 and \$9 for temporary storage of intermediate values

i=0 i<10

```
        add     $4, $0, $0          :  set $4=0 : 0 → i
loop:   slti    $8, $4, 10          : set $8=1 if $4 < 10 otherwise $8=0
        beq     $8, $0, end         : if $8=0 ($4>=10) branch to end label
        lui     $8, 0x1234          : $8 ← 0x12340000
        ori     $8, $8, 0x5678      : $8 ←$8 | 0x5678  : $8 =0x12345678
        sll     $9, $4, 2           : $9 ← $4 << 2  : $9 ←i*4
        add     $8, $8, $9          : form address of array[i] in $8
        sw      $0, 0($8)           : store 32-bits of zero from $0 into array[i]
i++     addui   $4, $4, 1           : i++
        beq     $0, $0, loop        : branch to label loop - always branches
end:
```



array
0
0
0
0
0
0
0
0
0
0
0x12345678
0x12345678+0
0x12345678+4
0x12345678+8
0x12345678+0xc
0x12345678+0x10
0x12345678+0x14
0x12345678+0x18
0x12345678+0x1C
0x12345678+0x20
0x12345678+0x24

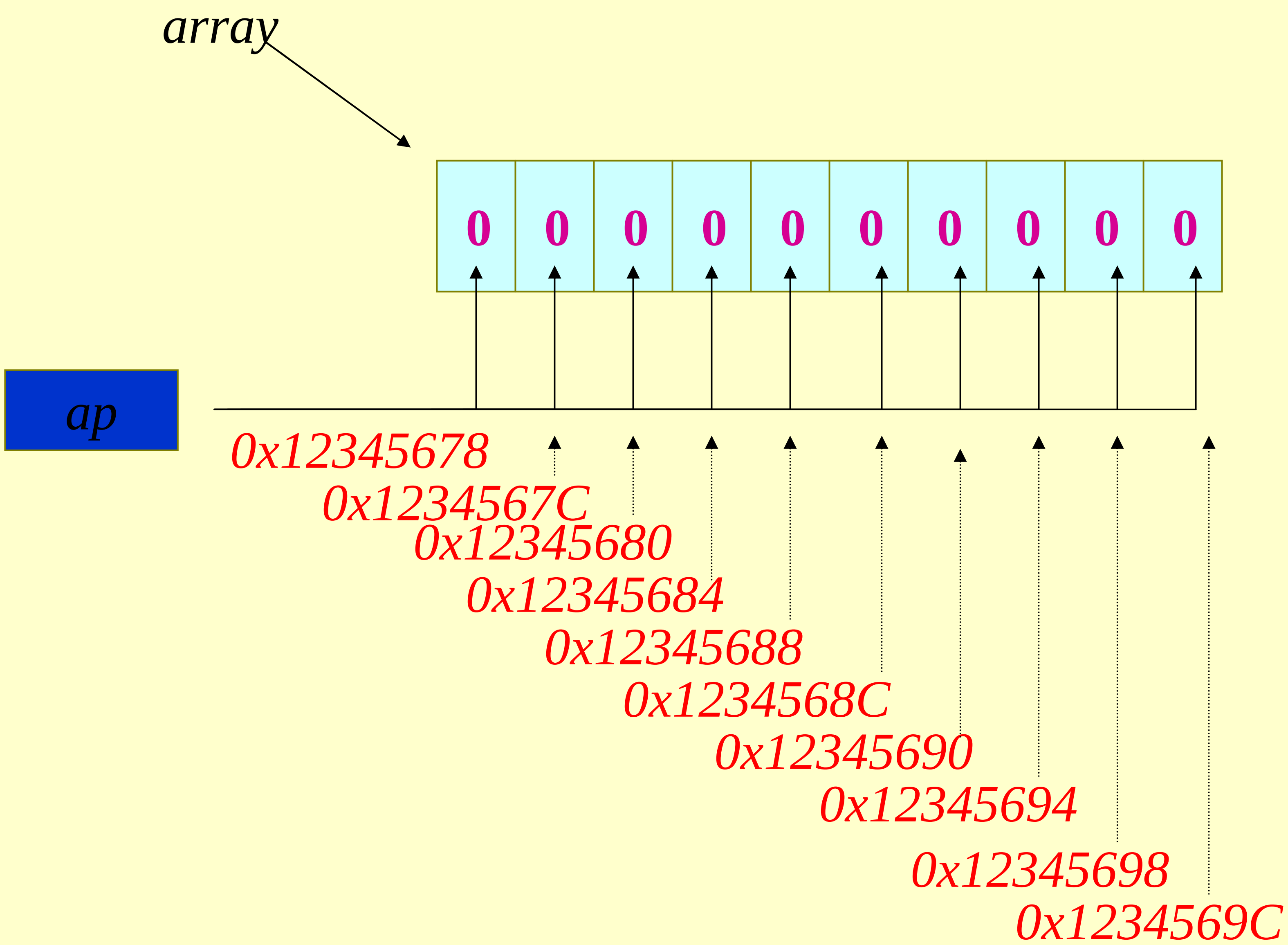
MIPS ***'for loop'*** *example*

Setting the elements of an array to zero,
but using pointers to memory addresses!
Data declarations (C code – ***NOT Java!!!***):-

```
unsigned i ;
int array[10] ;
int *ap ;
```

Variable '*ap*' is of *type* 'pointer to integer' and will hold an address (a pointer in C)

```
ap = array ;            // put the address of array into ap
for (i=0; i<10; i++) {
        *ap = 0 ;   // store 0 in the location pointed to by ap
        ap++ ;      // increment the address in ap by 4
                    // ap now points at the next element of array
}
```


array
0
0
0
0
0
0
0
0
0
0
ap
0x12345678
0x1234567C
0x12345680
0x12345684
0x12345688
0x1234568C
0x12345690
0x12345694
0x12345698
0x1234569C

MIPS 'for loop' *example*

Let the variable *i* be stored in register \$4, and variable *ap* in \$6
Let 'array' of integers be stored at address 12345678_{16}

Use \$8 for temporary storage

ap = array

```
        lui     $6, 0x1234          : $6 <- 0x12340000
        ori     $6, $6, 0x5678      : $6 <- $6 | 0x5678  : $6 =0x12345678

        add     $4, $0, $0          :  set  $4=0 : 0 → i
loop:   slti    $8, $4, 10          : set $8=1 if $4 < 10 otherwise 0
        beq     $8, $0, end         : if $8=0 ($4>=10) branch to end label
        sw      $0, 0($6)           : store 32-bits of zero in $0 into array[i]

        addui   $6, $6, 4           : ap++; add4 to $6 to point to array[i+1]
        addui   $4, $4, 1           : i++

        beq     $0, $0, loop        : branch to label loop - always branches
end:
```

Other instructions that change the PC:

jump register : **jr rs** : pc <- rs : register contents into pc

Register value must be multiple of 4 (or processor stops)
pc can be set to anywhere in memory (greater range than branches).
This is used to perform function return, e.g. jr $31,

N.B. Jumps can go a greater distance than branches.
However jumps are never conditional unlike branches.
Both are therefore necessary.

jump and link register : **jalr rs, rd** : rd <- pc ; **pc <- rs**

pc saved to register rd ***and then*** rs written into pc

Used for function (method) calls to **anywhere in the address space.**

Function (Method or Subroutine) Call

Some lines of program

0x0001AB2C _____________

0x0001AB30 _____________

0x0001AB34 **lui $1, 0x04** ***; $1 <- 0x00040000***

0x0001AB38 **ori $1, 0x5678** ***; $1 <- 0x00045678***

0x0001AB3C ***jalr $1, $31*** *; pc -> $31, $1->PC*

i.e.0x0001AB40->$31

0x00045678 -> PC

Function call

0x0001AB40 _____________

Function return

Code of method

0x00045678 _____________

0x0004567C _____________

0x00045680 _____________

0x00045684 _____________

0x00045688 jr $31 *; $31->PC*

i.e.0x01AB40 -> PC

Jump Instructions - *J Format*

6 Bits **op**	26 Bits **target**

jump to target : **j target** : pc[bits 27:0] ← target*4

jump and link target : **jal target**

register 31 <- contents of pc ; pc[bits 27:0] ← target*4

In both cases lower 28 bits of PC register are loaded with (26 bits of target field * 4)

jal is a method call instruction saving the PC before changing it.

Detail : pc is always an integer multiple of 4: therefore value stored in target field of instruction for j and jal is target address divided by 4, i.e. least 2 bits are dropped, since they are always 00.

Note: the upper 4-bits of PC are unchanged by these instructions.