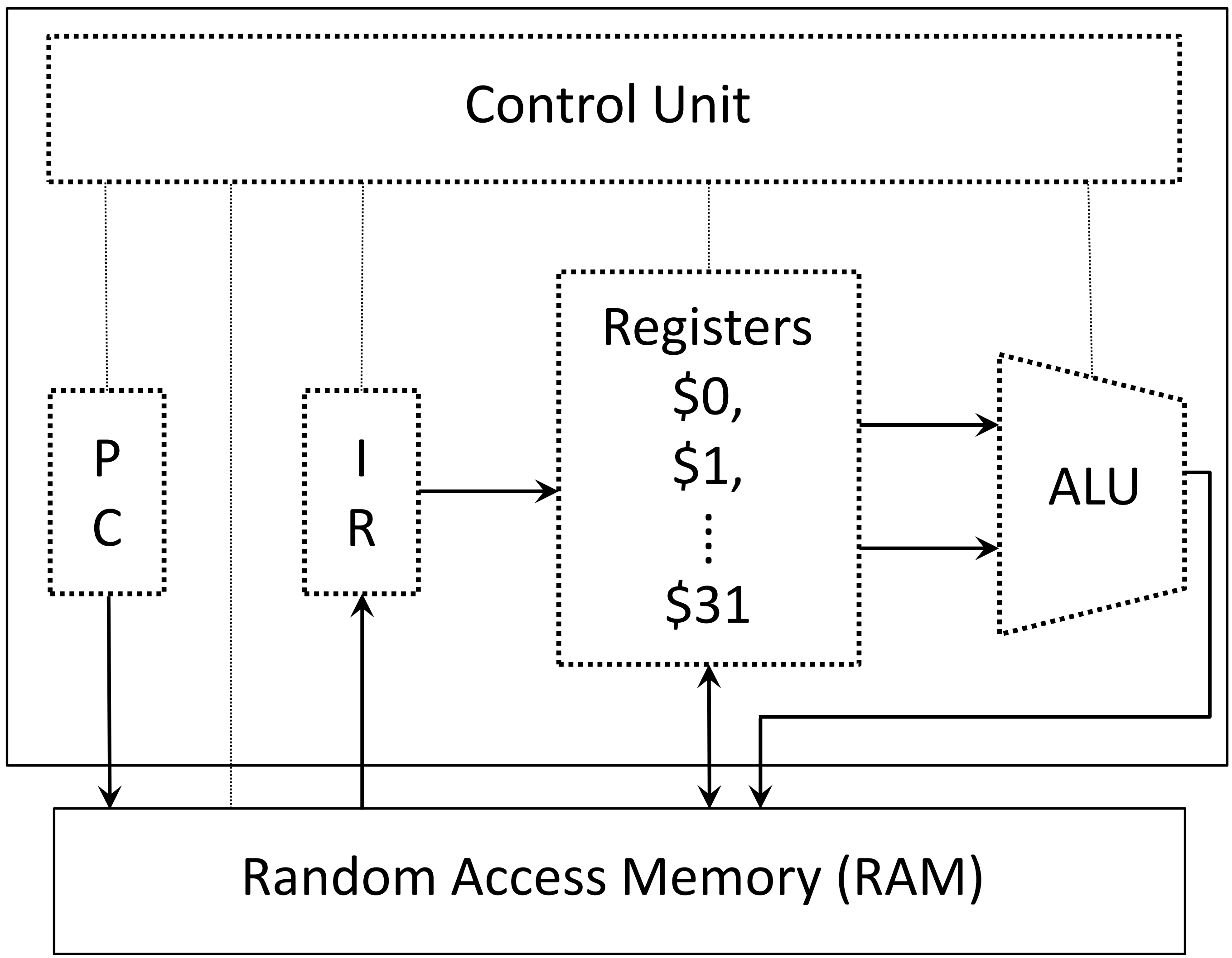
Simplified View of Processor and RAM



Simplified View of a Computer

Random Access Memory (RAM)

- *stores data (while power is on)*
- also called primary storage or main memory
- the processor can *directly access* literally billions of memory locations with instructions like load word and store word

Processor

- *manipulates data*
- consists of two part
  1. *control unit:* controls the flow of data throughout the processor
  2. *data path:* manipulates or processes the data

Simplified View of a Computer

Data Path

Major components include

- *Program Counter (PC):* holds the address of the current (or next) instruction
- *Instruction Register (IR):* holds the instruction that is being (or is about to be) executed
- *Arithmetic Logic Unit (ALU)*: performs arithmetic and logic calculations (add, sub, mult, div, and, or, not)
- *general purpose registers:* temporary (and fast) storage within the processor

Simplified View of a Computer

Missing from diagram …

Secondary Storage

- *stores data (even when power is off)*
- typically a hard disk drive (HDD), a solid state drive (SSD), or some combination of both
- not considered at this point

Input / Output Devices

- varies, but typically includes items such as a keyboard, mouse, display, speakers, USB ports
- not considered at this point

Conditional Execution

C++ vs. MIPS

- In general programming languages we need the ability to alter the path the computation takes depending on intermediate results
- in *C++* we have
  - if ... *then* ... else ...
  - while loops
  - for loops
- in *MIPS* we have
  - branch if equal (beq)
  - branch if not equal (bne)
  - set if less than (slt, sltu)

Conditional Execution

beq and bne

beq $s, $t, i

- branch if equal
- compare the contents of registers *$s* and *$t*
- *if equal*, skip *i* instructions
- *i* can be positive (to go forward) or negative (to go backwards)

bne $s, $t, i

- branch if not equal
- compare the contents of registers *$s* and *$t*
- *if not equal*, skip *i* instructions
- *i* can be positive or negative

Conditional Branches *beq* and *bne*

The Program Counter

- for branching to make sense, we need to understand the concept of a *PC* (program counter)
- recall
  - the PC stores a memory location i.e. it keeps track of where you are in the program
  - for MIPS, each instruction is 4 bytes long
  - machine code is stored in memory
- to run a program: get an instruction from memory, execute it, get the next instruction, execute it, ...
- *key point:* to get the next instruction, the processor increments the program counter by 4

Conditional Branches *beq* and *bne*

The Program Counter

- for example in the following code, assume *PC*=1000

Address	Contents
0x1000	add $4, $0, $0
0x1004	add $4, $4, $1
0x1008	add $4, $4, $2
0x100c	add $4, $4, $3

- The first instruction would be read in from memory location 1000 and executed. The *PC* would be incremented to 1004.
- The next instruction would be read in from memory location 1004 and executed. The *PC* would be incremented to 1008 ...
- *key point:* incrementing the *PC* happens automatically after each instruction is loaded into the Instruction Register (IR)

Conditional Branches *beq* and *bne*

The Program Counter

- to *skip over* some code (say skipping over one of the branches in an *if-then-else* statement) then *add a multiple of 4* to the PC
- to *skip back* in the code (say to go back to the beginning of a *for* or a *while* loop) *subtract off some multiple of 4* from the PC
- to start executing a specific subroutine, set the PC to the address where that subroutine starts
- in MIPS all addresses (of both data and code) are divisible by 4, i.e. in hexadecimal the address would always end in either 0, 4, 8 or c.

Conditional Branches *beq* and *bne*

Calculating how far to branch

- reference sheet definition
  bne $s, $t, i
  if ($s != $t) PC += i × 4
- this is saying if the contents of $s is not equal to the contents of $t then increment the counter by 4×i
- the size of each instruction is 4 bytes, so PC = PC + (i × 4) skips i instructions
- *key point:* this is in addition to regular PC increment by 4 that happens each time an instruction gets executed
- *final calc:* PC = PC + 4 + (i × 4)

Conditional Branches *beq* and *bne*

Calculating how far to branch

Addr	*Instruction*		
0x0ff8	sub $4, $4, $1	⟵	to go here *i* = -3
0x0ffc	sub $4, $4, $2	⟵	to go here *i* = -2
0x1000	beq $4, $5, i	⟵	to go here (*I* = -1) is an error
0x1004	add $4, $4, $3	⟵	happens anyway
0x1008	add $4, $4, $4	⟵	to go here *i* = 1
0x100c	add $4, $4, $5	⟵	to go here *i* = 2
0x1010	add $4, $4, $6	⟵	to go here *i* = 3

Conditional Setting

Set if Less Than (slt)

- useful if you don't want to test for equality but want to *test if the contents of one register is less than another*
- here *set* means make equal to 1 (or *True*)
- details

  slt $d, $s, $t

  compare register $s and $t

  if $s < $t then set $d (i.e. set $d = 1)

  if $s ≥ $t then reset $d (i.e. $d = 0)
- often it is used before **beq** and **bne**

Conditional Setting

Set if Less Than (slt)

- by reversing the order or the registers \$s and \$t in the slt instruction, i.e.

  slt \$d, \$s, \$t   vs.   slt \$d, \$t, \$s

  and combining with bne or beq we get 4 combinations

slt \$d, \$s, \$t bne \$d, \$0, i	slt \$d, \$s, \$t beq \$d, \$0, i
slt \$d, \$t \$s bne \$d, \$0, i	slt \$d, \$t, \$s beq \$d, \$0, i

- with these 4 combinations you can branch when:
  \$s < \$t, \$s ≤ \$t, \$s > \$t, or \$s ≥ \$t

Conditional Setting

Set if Less Than Unsigned (sltu)

- *many instructions which have integers as arguments come in two varieties: signed* and *unsigned*
- unsigned in another way of saying “natural numbers” where here natural numbers include 0
  - typically used for addresses
- signed is another way of saying “integers”
  - negative integers are represented using two’s complement
- with 32-bit architecture
  - unsigned have a range from 0 to $(2^{32} -1)$
  - signed have a range -2^{31} to $(2^{31}-1)$

Memory Model

Memory Access

- maximum size of memory : 2^{32} bytes = 4 GB
- think of it as one big array, *Mem*[ ]
- two different approaches to accessing memory
  - *byte addressing*:
    can access any of the 2^{32} bytes directly
  - *word aligned addressing*:
    can only access any of the 2^{30} words directly
    addresses must be divisible by 4
    only addresses 0, 4, 8, c, 10, 14,... are valid
    recall, 4 bytes in a word (for MIPS32)
- *MIPS uses word aligned addressing*

Memory Model

Memory Access

Note register *s* contains the address / location in memory where the item is to come from or go to.

```
lw $t, i($s)
```

- *load word* from Mem[$s + i] into *t*
- $s + i must be word-aligned (divisible by 4)

```
sw $t, i($s)
```

- *store word* from *t* into Mem[$s + i]
- $s + i must be word-aligned (divisible by 4)

Why immediate parameter *i* ? ...

Structures

Accessing Structures

- We have a parameter *i* because often many related items are stored in sequence (beside each other)
- *This parameter helps access each of the related items* in one instruction
  - local variables
  - accessing arguments in a function call

```
record_date (int year, int month, int day){
   …
}
```

Structures

Accessing Structures

```
record_date (int year, int month, int day){
  ...
```

- assume args are stored starting at address $1, to access...
  - the year: `lw $t, 0($1)`
  - the month: `lw $t, 4($1)`
  - the day: `lw $t, 8($1)`
- you must know the size of each item you are accessing
- what you are really saying is to access the ...
  - year add 0 to the address stored in register 1
  - month add 4 to the address stored in register 1
  - day add 8 to the address stored in register 1

More Arithmetic Operations in MIPS

Multiplication and Division

- these operations use special registers *hi, lo*

```
mult $s, $t
```

- multiply the contents of registers 's' and 't'
- result may be too big to fit in one register
- place most significant 32 bits in *hi*
- place least significant 32 bits in *lo*

```
div $s, $t
```

- divide the contents of register 's' by the contents of register 't ' and place result in *lo*, remainder in *hi*

More Arithmetic Operations in MIPS

Multiplication and Division

- *there are two versions of integers*
  - *unsigned: positive integers and 0 only*
  - *signed: positive and negative integers, i.e. two's complement*

`multu $s, $t`

- same as mult but treat the numbers in $s and $t as unsigned integers

`divu $s, $t`

- same as div but treat the numbers in $s and $t as unsigned integers

More Arithmetic Operations in MIPS

Accessing Results

- you gain access to the values stored in the special registers *hi* and *lo* using the **`mfhi`** and **`mflo`** commands

`mfhi $d`

- copy contents of hi to $d

`mflo $d`

- copy contents of lo to $d

Example of Arithmetic Operations in MIPS

Example: compute the average of three numbers

- values in $3, $4, $5
- place result in $2

```
 add $2, $3, $4          ; sum goes in $2
 add $2, $2, $5
 lis $1                  ; load in constant 3
.word 3
 div $2, $1              ; divide sum by 3
 mflo $2                 ; move answer to $2
```

Example of Conditional Branches

Example: Calculating 10 factorial

```
r2 = 1; // answer
r3 = 10;
while (r3 != 0) {
   r2 = r2 * r3;
   r3 -= 1;
}
```

- continue to *branch* back to the beginning of the loop, while r3 is *not equal* to 0
- the loop is based on the MIPS instruction *branch not equal* or *bne*

Example of Conditional Branches

Recall: Calculating 10 factorial

In C++ there are three parts:

1. initialize the variables
2. the while loop
3. the rest of code

```
r1 = 1;                    // 1. initialize
r2 = 1;                    //    answer
r3 = 10;
while (r3 != 0) {          // 2. while loop
   r2 = r2 * r3;
   r3 = r3 - r1;           // subtract 1
} ...                      // 3. rest of code
```

Example of Conditional Branches

In MIPS…

First *initialize our registers*

Address Contents Comments

```
                                   ; 1. initialize
0x00    lis $1                     ;      r1 = 1
0x04    .word 1                    ;

0x08    add $2, $1, $0             ;      r2 = 1

0x0C    lis $3                     ;      r3 = 10
0x10    .word 10                   ;
```

Example of Conditional Branches

In MIPS…

Now *implement the while loop*

Address Contents Comments

```
                                ; 2. while loop
0x14    mult $2, $3             ;    lo = r2*r3
0x18    mflo $2                 ;    r2 = lo
0x1A    sub  $3, $3, $1         ;    r3 = r3-1
0x20    bne  $3, $0, -4         ;
0x24    ...                     ; 3. rest of code
```

- `bne $3, $0, -4` means skip back 4 instructions if the contents of $1 is not equal to 0.

Example of Conditional Branches

In MIPS…

- **`bne $3, $0, -4`** means skip back 4 instructions if the contents of \$3 is not equal to 0.

- the actual calculation is as follows

- PC = 0x14 + 4 - 4*4 = 0x08

  0x14 location of the **`bne`** instruction

  +4 amount PC is incremented each time

  -4*4 the amount to subtract off the PC i.e. how far to skip back because of the **`bne`** instruction

Branch Labels

Calculating Offsets

- labels make assembly language easier: leave the computation of branch offsets to assembler
- *create a label* (single word followed by colon)
- *assembler program computes the actual offset*
- if you add more statement inside the loop, the assembler automatically recalculates the offset
- for assembly languages with variable length instructions, this is even more helpful

Branch Labels

Without Labels

Contents	Comments
	; 2. while loop
mult $2, $3	; lo = r2*r3
mflo $2	; r2 = lo
sub $3, $3, $1	; r3 = r3-1
bne $3, $0, -4	;

Branch Labels

With Labels

Contents | Comments

```
                                  ; 2. while loop
loop:  mult $2, $3                ;    lo = r2*r3
       mflo $2                    ;    r2 = lo
       sub  $3, $3, $1            ;    r3 = r3-1
       bne  $3, $0, loop          ;
```

`loop:` is the label

it is placed in first column and it ends with a colon