Chapter1. Introduction

2019年2月18日 14:06

Note Taking Area

Course Information

- Major in projects and designs, 后半学期比较忙.
- Grade percentage as listed.
  - Final exam: 35%
  - Midterm exam: 35%
  - Assignments: 5%
  - Lab part: 25%
  - 考试占比70%，请自裁。

Focus aspects

- CPU performance, memory management, multi-thread management.

Components

- “片子”和“板子”:
  - Chips: integrated circuit.
  - PCB board: traditional circuit.
- Input device, output device, memory, and central processor unit.
  - Memory: cache and flash memory.
  - In the CPU: datapath (performs operations on data), control (control the sequence of datapath, memory, I/O), and cache memory.
  - SRAM: Static RAM, save data into, random access, and then needn't to flash.
    - DRAM: dynamic RAM, flash and random access.
    - DISK: use pointer and doesn't random access.
  - Cache: 一级缓存、二级缓存先后找，最后找内存（非常数时间读取）。

Moore's Law and some key words

- Microprocessor advances: the number of transistors that can be integrated on a die would double every 18 to 24 months.
  - Transistor scale, the length decreases year by year.
- Clock frequency（时钟频率）: every second the clock changes its electrical level.

Between your program and hardware

- Application software: written in high-level language (HLL).
- System software:
  - Compiler: translates HLL code to machine code.
  - Operating system: service code, handling I/O, managing memory and storage, scheduling tasks & sharing resources.
- Hardware: processor, memory, I/O controllers.
- Levels of program code: HLL -> assembly language -> hardware representation.

Cue Column

Book related

- 计算机体系结构：expand one chapter into a whole book.
- 编码的奥秘、编码——隐匿在计算机软硬件背后：easy going book.

- 深入理解计算机系统：The courses related, traditional textbook.

How to do assembly programming

- Software: Qtspim or mars.
  - Assembly source file suffix: .s or .asm.
- System calls:

Service	System call code	Arguments	Result
print_int	1	$a0 = integer	
print_float	2	$f12 = float	
print_double	3	$f12 = double	
print_string	4	$a0 = string	
read_int	5		integer (in $v0)
read_float	6		float (in $f0)
read_double	7		double (in $f0)
read_string	8	$a0 = buffer, $a1 = length	
sbrk	9	$a0 = amount	address (in $v0)
exit	10		
print_char	11	$a0 = char	
read_char	12		char (in $v0)
open	13	$a0 = filename (string), $a1 = flags, $a2 = mode	file descriptor (in $v0)
read	14	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars read (in $v0)
write	15	$a0 = file descriptor, $a1 = buffer, $a2 = length	num chars written (in $v0)
close	16	$a0 = file descriptor	
exit2	17	$a0 = result	

- An example assembly source file:

```
.text
.globl main
main:
    la $a0, str
    li $v0, 4
    syscall
    li $v0, 10
    syscall
.data
str:    .asciiz "hello Internet\n"
```

  - In which is separated by ".text" and ".data", which includes codes and data.
  - In ".text", should be contains ".globl main".
  - Before system call "syscall", we should assign "la" and "li".
  - In ".data", if assign a "str" variable, should assign encoding type, which is ".asciiz".

Summaries

1. Course information.
2. Components.
3. Moore's Law.
4. Program and hardware.

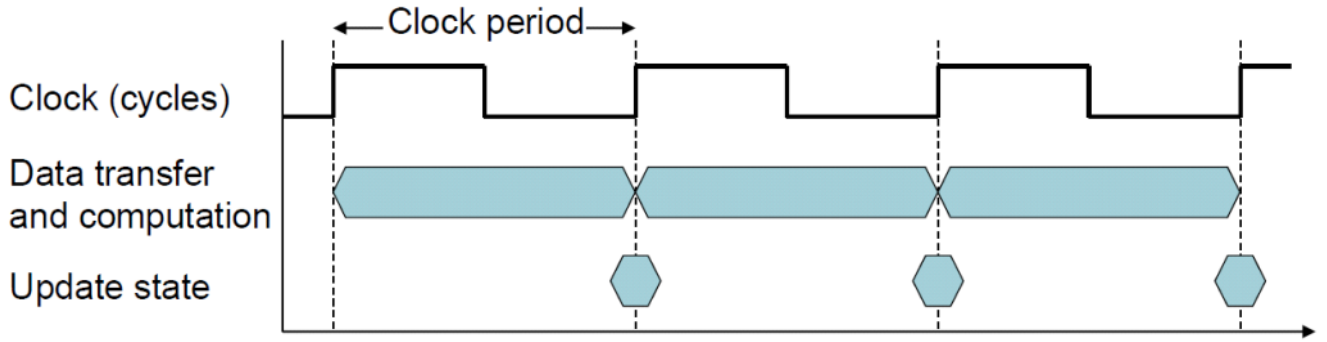
Chapter2. Performance

2019年2月25日 14:08

NOTE TAKING AREA

Definition performance

- Response time (time cost to do a work), throughput (total work done per unit time).
- Relative performance
  - Definition **performance**: P = 1 / Execution time.
  - X is **n times faster** than Y: $P_x / P_Y = n$.
- Measuring **execution time**:
  - **Elapsed time**: total response time, including all aspects, determines system performance.
  - **CPU time**: time spent processing a given job, user cpu time and system cpu time, different programs are affected differently by CPU and system performance.
- **CPU clocking**: operation of digital hardware governed by a constant-rate clock.



  - Clock period: duration of a clock cycle, 250ps.
  - Clock frequency (rate): cycles per second, 4.0GHz.
- **CPU time**: number of clock cycles * clock period = number / rate.
- Performance improved: reducing number of clock cycles, increase clock rate, hardware designer must often trade off clock rate against cycle count.

Instruction count and CPI

- CPI: cycles per instruction on average.
- **Clock cycles**: instruction counts * CPI.
- **CPU time**: instruction counts * CPI * clock cycle time = count * CPI / rate.
- Instruction count for a program: determined by program, ISA, and compiler.
  - ISA: instruction set architecture.
  - Average cycles per instruction: determined by CPU hardware, if different instructions have different CPI, then average CPI affected by instruction mix.
- CPU in more details:
  - If different instruction classes take different numbers of cycles:

$$\text{Clock Cycles} = \sum_{i=1}^{n} (\text{CPI}_i \times \text{Instruction Count}_i)$$

  - Weighted average CPI:

$$\text{CPI} = \frac{\text{Clock Cycles}}{\text{Instruction Count}} = \sum_{i=1}^{n} \left(\text{CPI}_i \times \frac{\text{Instruction Count}_i}{\text{Instruction Count}} \right)$$

    - Instruction count_i / instruction count = relative frequency.

Performance summary

- **CPU time = instructions / program = clock cycles / instruction = seconds / clock cycle.**
- Performance depends on algorithm (IC, possibly CPI), programming language (IC, CPI), compiler (IC, CPI), instruction set architecture (IC, CPI, T_C).
- **Power trends**: in CMOS IC technology, because of leakage of currency, there can be leak power in CPU, contains static power and dynamic power.
  - Static power = leak currency * voltage.
  - Dynamic power = capacitive load * voltage^2 * frequency.
  - Total power = static power + dynamic power.
- Uniprocessor performance: constrained by power, instruction-level parallelism, memory latency.
- Multiprocessor:
  - Multiple microprocessors: more than one processors per chip.
  - Requires explicitly parallel programming.

Amdahl's Law

- Architecture is very bottleneck-driven: make the common case fast.
- **Amdahl's Law**: performance improvements through an enhancement is limited by the fraction of time the enhancement comes into play.

  Example: multiply accounts for 80s/100s

  - How much improvement in multiply performance to get 5× overall?

    $$20 = \frac{80}{n} + 20$$

    - Can't be done!
  - Corollary: make the common case fast.
- Fallacy: low power at idle.

CUE COLUMN

CPU time example

Computer A: 2GHz clock, 10s CPU time

Designing Computer B

- Aim for 6s CPU time
- Can do faster clock, but causes 1.2 × clock cycles

How fast must Computer B clock be?

- The solution can be found by following steps:

$$\text{Clock Rate}_B = \frac{\text{Clock Cycles}_B}{\text{CPU Time}_B} = \frac{1.2 \times \text{Clock Cycles}_A}{6\text{s}}$$

$$\text{Clock Cycles}_A = \text{CPU Time}_A \times \text{Clock Rate}_A = 10\text{s} \times 2\text{GHz} = 20 \times 10^9$$

$$\text{Clock Rate}_B = \frac{1.2 \times 20 \times 10^9}{6\text{s}} = \frac{24 \times 10^9}{6\text{s}} = 4\text{GHz}$$

CPI example

Computer A: Cycle Time = 250ps, CPI = 2.0

Computer B: Cycle Time = 500ps, CPI = 1.2

Same ISA

Which is faster, and by how much?

- Same ISA means the same instruction set architecture, which means same instruction counts.

$$\text{CPU Time}_A = \text{Instruction Count} \times \text{CPI}_A \times \text{Cycle Time}_A$$
$$= I \times 2.0 \times 250\text{ps} = I \times 500\text{ps} \leftarrow \text{A is faster...}$$
$$\text{CPU Time}_B = \text{Instruction Count} \times \text{CPI}_B \times \text{Cycle Time}_B$$
$$= I \times 1.2 \times 500\text{ps} = I \times 600\text{ps}$$
$$\frac{\text{CPU Time}_B}{\text{CPU Time}_A} = \frac{I \times 600\text{ps}}{I \times 500\text{ps}} = 1.2 \leftarrow \text{...by this much}$$

Reducing power example

Suppose a new CPU has

- 85% of capacitive load of old CPU
- 15% voltage and 15% frequency reduction

$$\frac{P_{new}}{P_{old}} = \frac{C_{old} \times 0.85 \times (V_{old} \times 0.85)^2 \times F_{old} \times 0.85}{C_{old} \times V_{old}^{\ 2} \times F_{old}} = 0.85^4 = 0.52$$

Assemble program design

- Comment: start with '#', this line is comment.
- **Data declaration: basic style "name: storage_type value".**

```
example

var1:       .word   3       # create a single integer:
                            #variable with initial value 3

array1:     .byte   'a','b' # create a 2-element character
                            # array with elements initialized:
                            # to  a  and  b

array2:     .space  40      # allocate 40 consecutive bytes,
                            # with storage uninitialized
                            # could be used as a 40-element
                            # character array, or a
                            # 10-element integer array;
                            # a comment should indicate it.

string1     .asciiz "Print this.\n"         #declare a string
```

  - One word equals four bytes.
- Registers and their name:

寄存器名称	编号	使用规则	寄存器名称	编号	使用规则
$zero	0	恒为0	$s0	16	保存临时值（过程调用预留）
$at	1	为汇编器保留	$s1	17	保存临时值（过程调用预留）
$v0	2	表达式求值以及函数的结果	$s2	18	保存临时值（过程调用预留）
$v1	3	表达式求值以及函数的结果	$s3	19	保存临时值（过程调用预留）
$a0	4	参数1	$s4	20	保存临时值（过程调用预留）
$a1	5	参数2	$s5	21	保存临时值（过程调用预留）
$a2	6	参数3	$s6	22	保存临时值（过程调用预留）
$a3	7	参数4	$s7	23	保存临时值（过程调用预留）
$t0	8	临时（不为过程调用预留）	$t8	24	临时（不为过程调用预留）
$t1	9	临时（不为过程调用预留）	$t9	25	临时（不为过程调用预留）
$t2	10	临时（不为过程调用预留）	$k0	26	为OS内核保留
$t3	11	临时（不为过程调用预留）	$k1	27	为OS内核保留
$t4	12	临时（不为过程调用预留）	$gp	28	全局区域的指针
$t5	13	临时（不为过程调用预留）	$sp	29	堆栈指针
$t6	14	临时（不为过程调用预留）	$fp	30	帧指针
$t7	15	临时（不为过程调用预留）	$ra	31	返回地址（函数调用使用）

图 B-6-1　MIPS 寄存器和使用规则

- Load command: lw (load word), lb (load byte), li (load immediate value) - destination, RAM/value.
- Store command: sw (save word), sb (save byte) - source, destination.
- Memory addressing:
  - Load addressing: direct addressing the value.

```
la          $t0, var1
```

    - Put the address of var1 from ".data" into t0.
  - Indirect addressing: find address from register.

```
lw          $t2, ($t0)
```

- Based or indexed addressing: using offset addressing.

```
lw          $t2, 4($t0)
```

  - The front number is the offset, which can be positive or negative integers.

- Arithmetic instructions:

```
add         $t0,$t1,$t2     #  $t0 = $t1 + $t2;   add as signed
                            # (2's complement) integers
sub         $t2,$t3,$t4     #  $t2 = $t3 Ð $t4
addi        $t2,$t3, 5      #  $t2 = $t3 + 5;   "add immediate"
                            # (no sub immediate)
addu        $t1,$t6,$t7     #  $t1 = $t6 + $t7;
addu        $t1,$t6,5       #  $t1 = $t6 + 5;
                            #  add as unsigned integers
subu        $t1,$t6,$t7     #  $t1 = $t6 - $t7;
subu        $t1,$t6,5       #  $t1 = $t6 - 5
                            #  subtract as unsigned integers

mult        $t3,$t4         #  multiply 32-bit quantities in $t3
                            #  and $t4, and store 64-bit
                            #  result in special registers Lo
                            #  and Hi:  (Hi,Lo) = $t3 * $t4

div         $t5,$t6         #  Lo = $t5 / $t6   (integer quotient)
                            #  Hi = $t5 mod $t6   (remainder)
mfhi        $t0             #  move quantity in special register Hi
                            #  to $t0:   $t0 = Hi
mflo        $t1             #  move quantity in special register Lo
                            #  to $t1:   $t1 = Lo,  used to get at
                            #  result of product or quotient

move        $t2,$t3         #  $t2 = $t3
```

  - In which mflo is quotient, and mfhi is the reminder.

SUMMARIES

1. Definition performance, clock and cpu time.
2. Instruction count and CPI.
3. Performance summary.
4. Amdahl's Law, corollary and fallacy.

Chapter3. Instruction Set Architecture

2019年3月5日 8:18

NOTE TAKING AREA

Instruction set architecture

- Instruction set
  - Hardware language:
    - Instructions: words of a computer's language.
    - Instruction set: vocabulary of commands.
  - Two form of instruction set:
    - Assembly language: written by people.
    - Machine language: read by computer.
  - Instruction set of different machine are similar.
  - Design target: easy to build, maximizing performance.
  - Important design principles:
    - Keep the hardware simple.
    - Keep the instruction regular.
- RISC / CISC: reduced instruction set computer / complex instruction set computer.
- **Design principle**
  - **Simplicity favors regulars.**
    i. **Regularity** makes implementation simpler.
    ii. **Simplicity** enables higher performance at lower cost.
  - **Smaller is fast.**
  - **Make the common case fast.**
  - **Good design demands good compromises.**
    - Different formats complicate decoding, but allow 32-bit instructions uniformly.
    - Keep formats as similar as possible.
- MIPS / ARM / x86

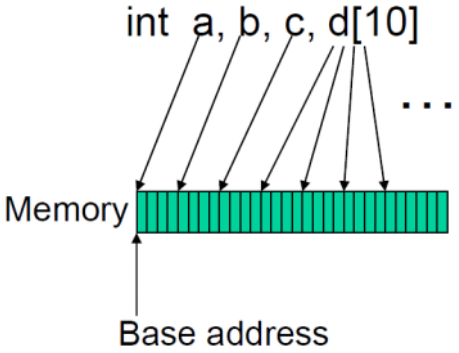
Instructions

- Arithmetic instruction.
- Data transfer instruction.
- Logical instruction:
  - Shift left: sll.
  - Shift right: srl.
  - Bit-by-bit and: and, andi.
  - Bit-by-bit or: or, ori.
  - Bit-by-bit not or: nor.
- Conditional branch:
  - Jump to instruction L1 if register1 equals register2: beq register1, register2, L1.
    - Similarly, bne and slt (set on less than).
  - Unconditional branch: j L1 / jr $s0.

Basic concepts

- Operands: register / memory / immediate.
  - Word: a 32-bit entity (4 bytes).
  - Values must be fetched from memory before instructions can operate on

them.
    - Memory address: store the location of every variables.



    - **Memory address is in unit of byte.**
  - Registers and memory:
    i. Registers are faster to access than memory.
    ii. Operating on memory data requires loads and stores.
    iii. Compiler must use registers for variables as much as possible.
- Numeric representation: signed / unsigned / sign extension.
  - Unsigned binary integers: $X = X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + \ldots + X_1 2^1 + X_0 2^0$.
    - Range from 0 to 2^n-1. For 32 bit integer, it's 0 to +4,294,967,295.
  - 2s-complement signed integer: $X = -X_{n-1}2^{n-1} + X_{n-2}2^{n-2} + \ldots + X_1 2^1 + X_0 2^0$.
    - Range from -2^{n-1} to $+2^{n-1}$-1. For 32 bit integer, from -2,147,483,648 to 2,147,483,647.
  - 1s-complement: equals 2's complement -1, called radix-minus-one complement.
    - Simply change 0 to 1 and 1 to 0.
  - Extend 8 bits to 16 or 32 bits: add sign bit to the left, and right shift the rest bits.
    - Example: 0000 0001 -> 0000 0000 0000 0001.
    - Example: 1111 1101 -> 1111 1111 1111 1101.
- Instruction format: R-format / I-format
  - Instructions are represented as 32-bit numbers, broke into 6 fields.
  - R-type instruction: add $t0, $s1, $s2.

000000	10001	10010	01000	00000	000000
6 bits	5 bits	5 bits	5 bits	5 bits	6 bits
op	rs	rt	rd	shamt	funct
opcode	source	source	dest	shift amt	function

  - I-type instruction: lw $t0, 32($s3)

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	constant

CUE COLUMN

Big-endian and small-endian

- Big-endian: the most-significant in the lower address in memory.
  - 0x12345678 in memory:

0	4	8	12
0x12	0x34	0x56	0x78

- Small-endian: the less-significant in the higher address in memory.
  - 0x12345678 in memory:

0	4	8	12
0x78	0x56	0x34	0x12

Logic operation and shift operation

- Login operation

Description	Op-code	Operand
Add with Overflow	add	destination, src1, src2
Add without Overflow	addu	destination, src1, src2
AND	and	destination, src1, immediate
Divide Signed	div	destination/src1, immediate
Divide Unsigned	divu	
Exclusive-OR	xor	
Multiply	mul	
Multiply with Overflow	mulo	
Multiply with Overflow Unsigned	mulou	
NOT OR	nor	
OR	or	
Set Equal	seq	
Set Greater	sgt	
Set Greater/Equal	sge	
Set Greater/Equal Unsigned	sgeu	
Set Greater Unsigned	sgtu	
Set Less	slt	
Set Less/Equal	sle	
Set Less/Equal Unsigned	sleu	
Set Less Unsigned	sltu	
Set Not Equal	sne	
Subtract with Overflow	sub	
Subtract without Overflow	subu	

- Shift operation

Description	Op-code	Operand
Rotate Left	rol	
Rotate Right	ror	
Shift Right Arithmetic	sra	
Shift Left Logical	sll	
Shift Right Logical	srl	
Absolute Value	abs	destination,src1
Negate with Overflow	neg	destination/src1
Negate without Overflow	negu	
NOT	not	
Move	move	destination,src1
Multiply	mult	src1,src2
Multiply Unsigned	multu	

  - Details of shift operations:

sll	Shift left logical	Shift left and insert zero at the least-significant bit.
sra	Shift right arithmetic	Insert the sign bit at the most-significant bit.
srl	Shift right logical	Insert zero at the most-significant bit.
rol	Rotate left	The instruction inserts in the least-significant bit any bits that were shifted out of the sign bit.
ror	Rotate right	Similar to rol but rotate add shifted out of least-significant bit.

SUMMARIES

1. Instruction set architecture.
2. Instructions.
3. Basic concepts.

Chapter4. Instruction Set Architecture 2

2019年3月11日 14:12

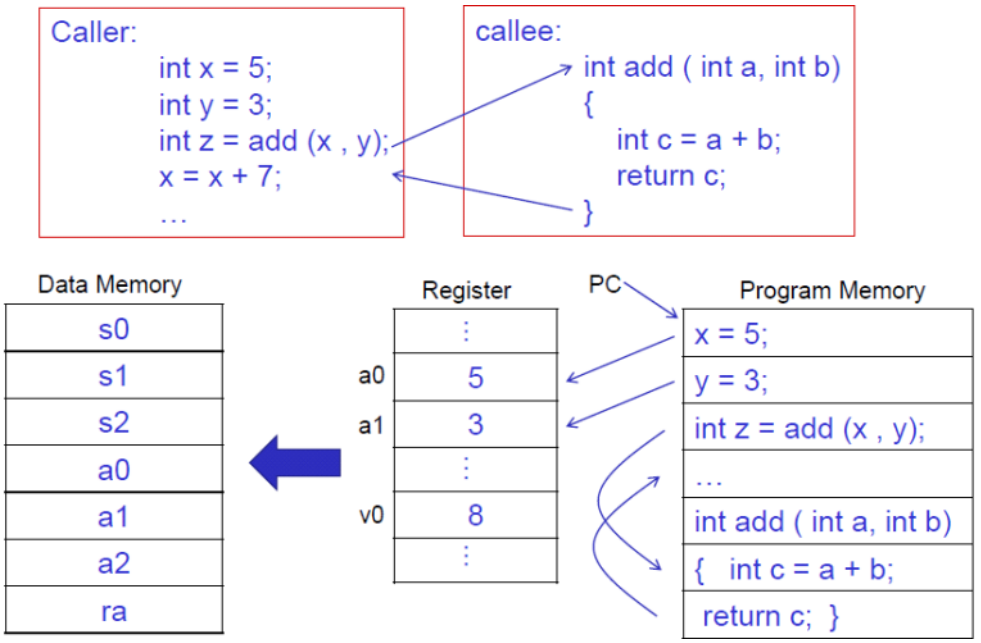
NOTE TAKING AREA

Control instructions: if else

- Condition branch: jump to instruction L1 if register1 equals register2: $beq\ register1, register2, L1$.
  - Unconditional branch: $j\ L1$, $jr\ \$s0$.
- More conditional operations:
  - $slt\ rd, rs, rt$: if (rs < rt) rd = 1 else rd = 0.
  - $slti\ rd, rs, constant$: if (rs < constant) rt = 1 else rd = 0.
  - $bne\ \$t0, \$zero, L$: if $t0 doesn't equal to zero, then jump to L.
- Pseudo instructions: there is **no** such instructions in hardware, the assembler **translates** them into a combination of real instructions.
  - blt $s0, $s1, Label
    - If s0<s1, jump to Label
  - bgt $s0, $s1, Label
    - If s0<s1, jump to Label
  - ble $s0, $s1, Label
    - If s0<=s1, jump to Label
  - beqz $s0, Label
    - If s0==0, jump to Label
  - li $t0, 5
    - Load immediate, t0 = 5
  - Move $t0, $s0
    - t0 = s0
  - Why not blt, bge or something?
    - Hardware for <, >= are slower than =, ≠.
- Signed and unsigned:
  - Signed comparison: slt, slti.
  - Unsigned comparison: sltu, sltui.

Procedures

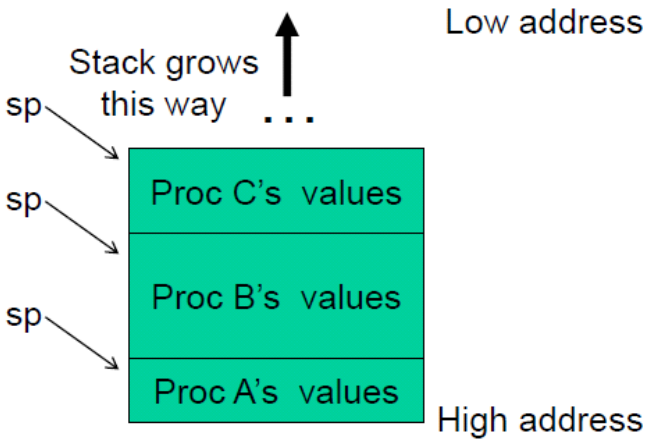
- Procedure/function: one tool used by the programmers to structure programs.
  - Easy to understand and reuse code.
- Steps to execute procedure:



  - Parameters (arguments) are places where the callee can see them.
  - Control is transferred to the callee.
  - Acquire storage resources for callee.
  - Execute the procedure.
  - Place result value where caller can access it.
  - Return control to caller.
- Registers used during procedure calling:
  - $a0 - $a3: four **argument registers** to pass parameters.
  - $v0 - $v1: two **value register to return** values.
  - $ra: one return **address register** to return to the point of origin in the caller.

Registers and stack

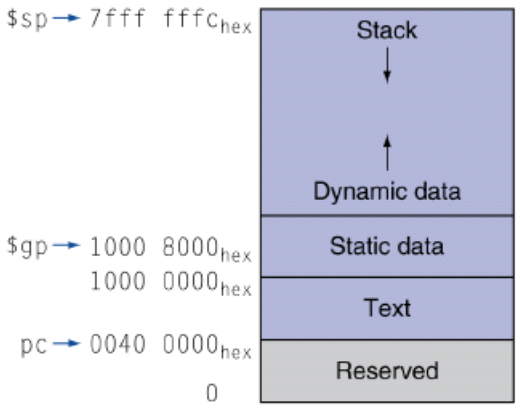
- Jump and link:
  - Program counter (PC): a special register maintains the address of the instruction currently being executed.
  - We use jal to reach procedure's address and the return address (PC), actually, PC+4 is stored in the $ra.
  - Avoid overwriting $ra, we need to store it first.
- Registers: 32 MIPS registers.
  - Register 0 : $zero always stores the constant 0
  - Regs 2-3 : $v0, $v1 return values of a procedure
  - Regs 4-7 : $a0-$a3 input arguments to a procedure
  - Regs 8-15 : $t0-$t7 temporaries
  - Regs 16-23: $s0-$s7 variables
  - Regs 24-25: $t8-$t9 more temporaries
  - Reg 28 : $gp global pointer
  - Reg 29 : $sp stack pointer
  - Reg 30 : $fp frame pointer
  - Reg 31 : $ra return address
- The stack: the registers for a procedure are volatile, it disappears every time we switch procedures. Then we need to store registers in stack.



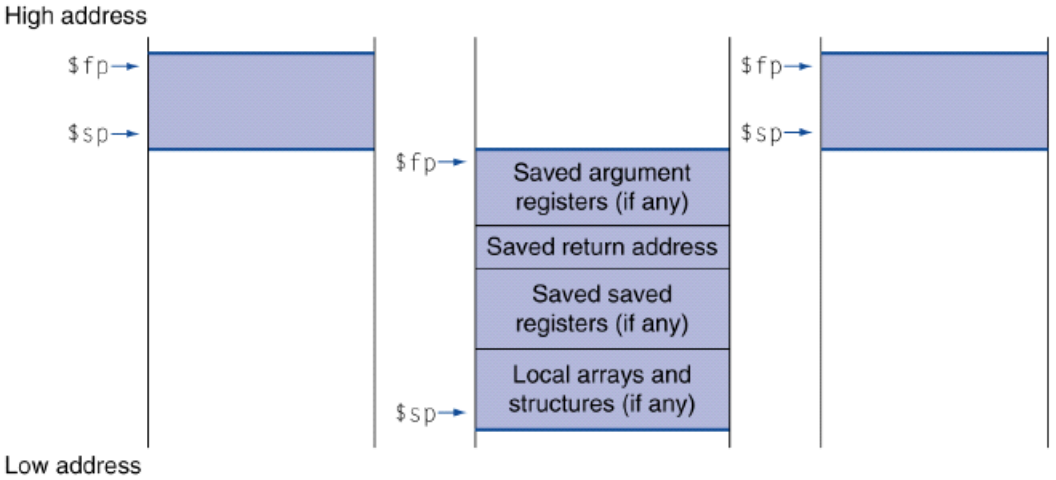

```
Proc  A

        call  Proc B
              …
              call Proc C
                 …
              return
        return
return
```

- Storage management on a call/return
  - *A new procedure must create space for all its variables on the stack.*
    - The registers need to storage: $s0 - $s7, $a0 - $a3, $ra.
  - The callee creates stack space, it updates the value of $sp.
  - The callee finished, all the values in stack copied back to register, and free stack memory.
- Memory layout:



  - Text: program code.
  - Static data: global variables, example static variables in C, constant arrays and strings.
    - $gp: initialized to address allowing ±offsets into this segment.
  - Dynamic data: heap, example new in Java.
  - Stack: automatic storage.
- Local data on the stack:



  - **Local data allocated by callee** (example C automatic variables) and **procedure frame** (activation record) - used by some compilers to manage stack storage.

CUE COLUMN

Conditional / unconditional branch

Convert to assembly:

```
if (i == j)
    f = g+h;
else
    f = g-h;
```

```
bne  $s3, $s4, Else
add  $s0, $s1, $s2
j    Exit
Else: sub  $s0, $s1, $s2
Exit:
```

Loop example

- Convert following code into assembly:

  $while\ (save[i] == k)$
  $i += 1;$

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit:
```

- Variable i and k are in $s3 and $s5 and base of array save[] is in $s6.

Example of leaf procedure

```
int  leaf_example (int g, int h, int i, int j)
{
   int f ;
   f = (g + h) – (i + j);
   return f;
}
```

The caller has saved:
g→$a0,
h→$a1,
i→$a2,
j→$a3,
return address → $ra

```
leaf_example:
addi   $sp,  $sp,  -12
sw     $t1, 8($sp)
sw     $t0, 4($sp)
sw     $s0, 0($sp)
add    $t0, $a0, $a1
add    $t1, $a2, $a3
sub    $s0, $t0, $t1
add    $v0, $s0, $zero
lw     $s0, 0($sp)
lw     $t0, 4($sp)
lw     $t1, 8($sp)
addi   $sp, $sp, 12
jr     $ra
```

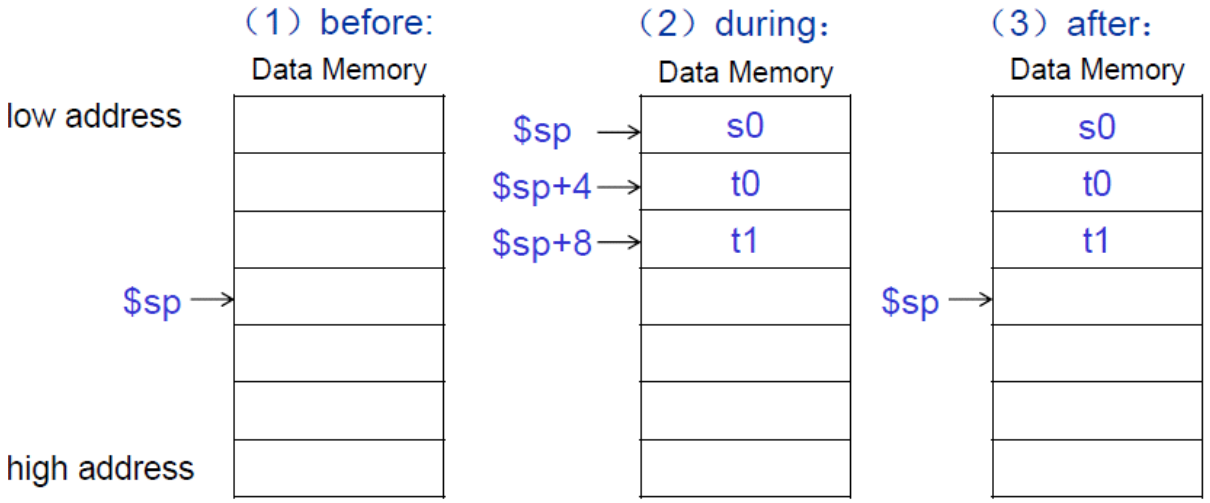
Save t0,t1,s0
Protect environment

Procedure body

Restore t0 t1 s0

Return result

- Note the usage of stack, from offset 8 to offset 0, so the stack memory increase from high address to low address.
- The corresponding data in stack:



- To avoid too many memory operations:
  - $t0 - $t9: temporary registers are not preserved by the callee.
  - $s0 - $s7: saved registers must be preserved by the callee if used..

Example of non-leaf procedure

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

```
fact:
    addi    $sp, $sp, -8
    sw      $ra, 4($sp)
    sw      $a0, 0($sp)
    slti    $t0, $a0, 1
    beq     $t0, $zero, L1
    addi    $v0, $zero, 1
    addi    $sp, $sp, 8
    jr      $ra
L1:
    addi    $a0, $a0, -1
    jal     fact
    lw      $a0, 0($sp)
    lw      $ra, 4($sp)
    addi    $sp, $sp, 8
    mul     $v0, $a0, $v0
    jr      $ra
```

Compare n<1

Return 1

Fact(n-1)

Return n*fact(n-1)

Notes:

The caller saves $a0 and $ra in its stack space.

Temps are never saved.

- For this condition, $ra and $a0 - $a3 are stored.

SUMMARIES

1. Control instructions for conditional and unconditional.
2. Procedures.
3. Registers and stack, memory layout.

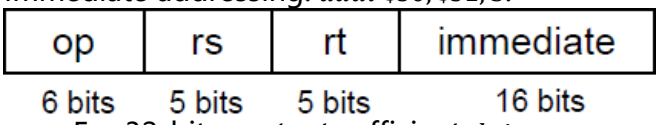
Chapter5. Instruction Set Architecture 3

2019年3月18日 14:40

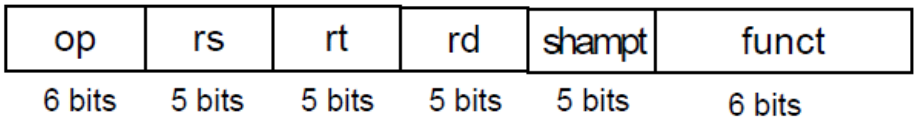
NOTE TAKING AREA

MIPS addressing

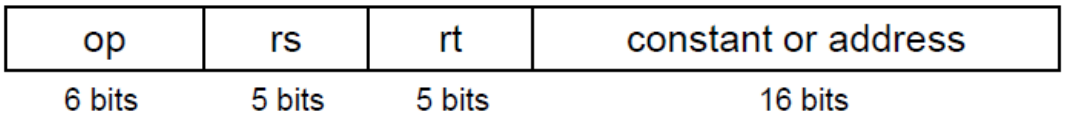
- Addressing: how the instructions identify the operands of the instructions.
- Immediate addressing: $addi\ \$s0, \$s1, 5$.



  - For 32-bit constant sufficient: $lui\ rt, constant$, copy 16-bit constant to left 16 bits of rt, and clears right 16 bits of rt to 0.
- Register addressing: $add\ \$s0, \$s1, \$s2$.



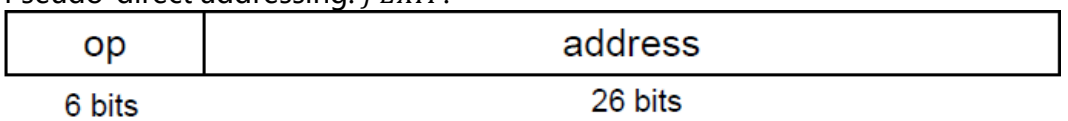
- Base / Displacement address: $lw\ \$s0, 0(\$s1)$.
  - Like the register addressing, but with a shift value.
- PC-relative addressing: $bne\ \$s0, \$s1, EXIT$.



  - Also called branch addressing, used in branch instructions.
  - Branching far away: the assembler will rewrite the code:

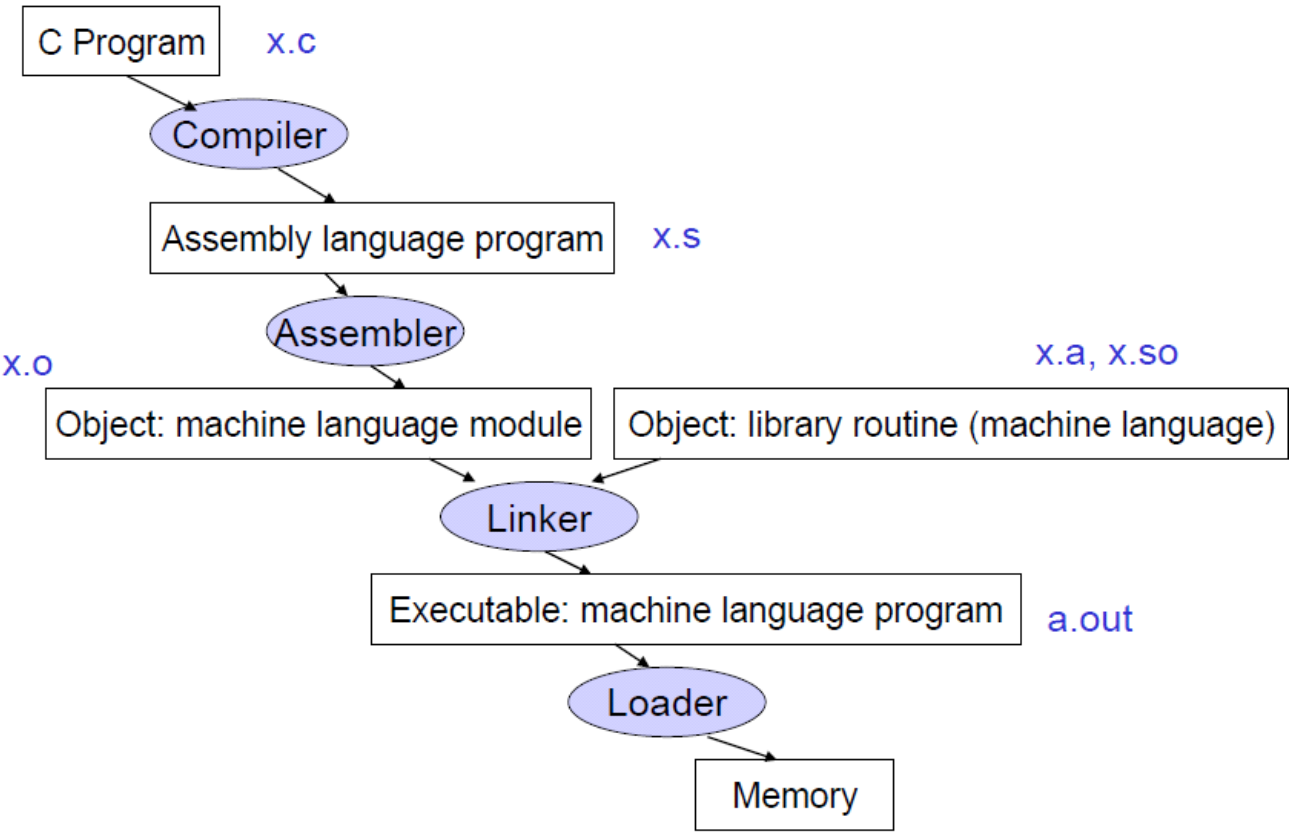
```
        beq $s0,$s1, L1
              ↓
        bne $s0,$s1, L2
        j L1
L2:     …
```

- Pseudo-direct addressing: $j\ EXIT$.

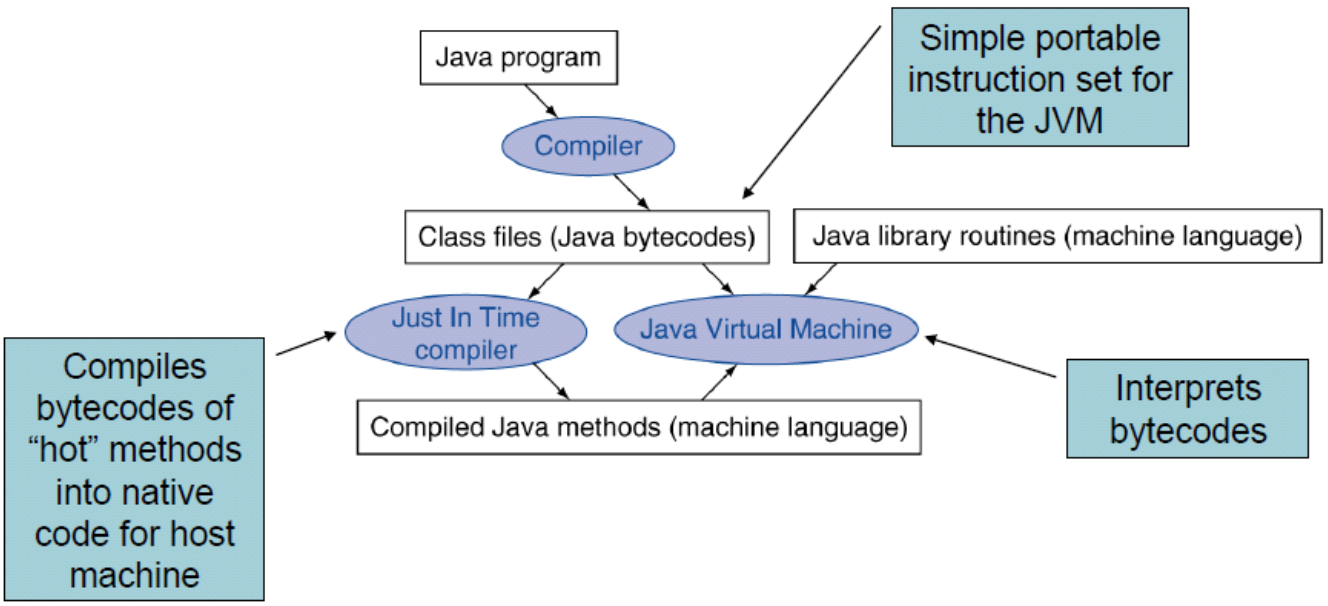


Translating and starting a program

- The steps of executing a C program:



  - Assembler: convert pseudo-instructions into actual hardware instructions, and convert assembly instrs into machine instrs.
  - Linker: stitches different object files into a single executable (patch internal and external references, determine addresses of data and instruction labels, and organize code and data modules in memory), some libraries are dynamically linked.
- The application of Java:



Other popular ISA

- ARM and x86, but I'll skip this part.

CUE COLUMN

Target addressing example

```
Loop: sll  $t1, $s3, 2
      add  $t1, $t1, $s6
      lw   $t0, 0($t1)
      bne  $t0, $s5, Exit
      addi $s3, $s3, 1
      j    Loop
Exit: …
```

80000	0	0	19	9	4	0
80004	0	9	22	9	0	32
80008	35	9	8	0		
80012	5	8	21	2		
80016	8	19	19	1		
80020	2	20000				
80024						

Decoding machine language table

- The table of op code:

op(31:26)							
28–26 31–29	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	R-format	Bltz/gez	jump	jump & link	branch eq	branch ne	blez	bgtz
1(001)	add immediate	addiu	set less than imm.	set less than imm. unsigned	andi	ori	xori	load upper immediate
2(010)	TLB	FlPt						
3(011)								
4(100)	load byte	load half	lwl	load word	load byte unsigned	load half unsigned	lwr	
5(101)	store byte	store half	swl	store word			swr	
6(110)	load linked word	lwc1						
7(111)	store cond. word	swc1						

- For op code is 000000, R-format condition, the funct code will be:

op(31:26)=000000 (R-format), funct(5:0)								
2–0 5–3	0(000)	1(001)	2(010)	3(011)	4(100)	5(101)	6(110)	7(111)
0(000)	shift left logical		shift right logical	sra	sllv		srlv	srav
1(001)	jump register	jalr			syscall	break		
2(010)	mfhi	mthi	mflo	mtlo				
3(011)	mult	multu	div	divu				
4(100)	add	addu	subtract	subu	and	or	xor	not or (nor)
5(101)			set l.t.	set l.t. unsigned				
6(110)								
7(111)								

SUMMARIES

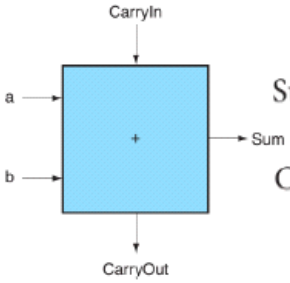
Chapter6. Arithmetic for Computers

2019年3月25日 14:16

NOTE TAKING AREA

Operations on integers

- Addition and subtraction
  - Addition overflow: positive and negative value addition won't have overflow. However, two positive or negative values addition can get overflow.
  - 1-bit adder:



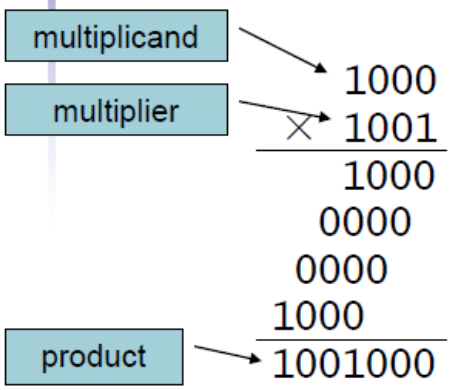
$$\text{Sum} = (a \cdot \bar{b} \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot b \cdot \overline{\text{CarryIn}}) + (\bar{a} \cdot \bar{b} \cdot \text{CarryIn}) + (a \cdot b \cdot \text{CarryIn})$$

$$\text{CarryOut} = (b \cdot \text{CarryIn}) + (a \cdot \text{CarryIn}) + (a \cdot b)$$

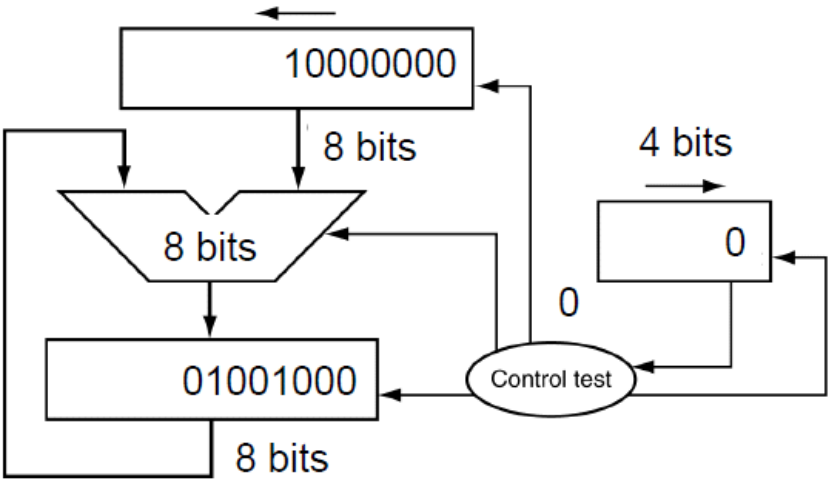
    - The component of 1-bit adder can be result in any bit adder.
  - Integer subtraction: two positive or negative values won't get overflow, but one positive and one negative numbers may get overflow.
- Dealing with overflow:
  - Languages ignore overflow: such as C, addu, addiu, subu in MIPS.
  - Languages raising an exception: add, addi, sub in MIPS.
    - Exception handler: save PC in exception program counter (EPC) register, and jump to predefined handler address, finally mfc0 instruction can retrieve EPC value.

Arithmetic for multimedia

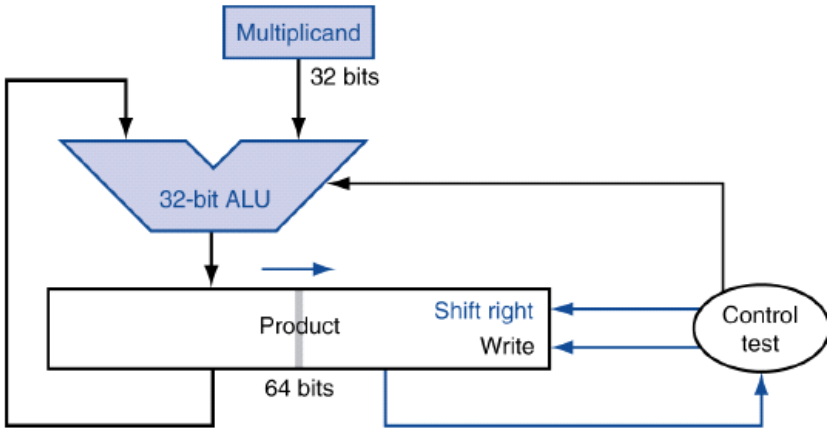
- 8 bit and 16 bit data vector will be operated on graphics and media processing.
- Use 64-bit adder with partitioned carry chain to process multimedia.
- If overflow, result will be set to the largest representable value.



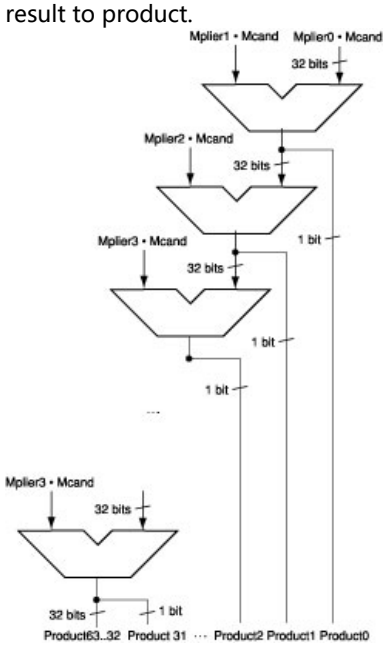
Length of product is the sum of operand lengths



  - In every step: multiplicand is shifted, and next bit of multiplier is examined, and shift multiplicand of 1 is added to the product.
- Optimized multiplier: steps in parallel, add and shift.



- Faster multiplier: instead requires a clock to ensure that the earlier addition has completed before shifting, process addition meanwhile send addition result to product.

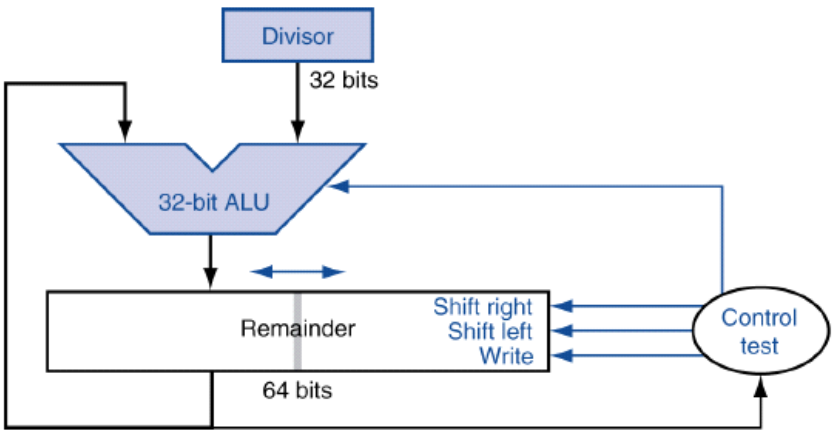


- Uses multiple pipelined adders, cost and performance tradeoff.

- MIPS multiplication: two 32-bit registers for product, HI for most-significant 32 bits and LO for least-significant 32 bits.
  - Instructions: $mult\ rs,rt$ | $multu\ rs,rt$ | $mfhi\ rd$ | $mflo\ rd$ | $mul\ rd,rs,rt$.

Division

- check for 0 division, for signed division, divide using absolute values, and adjust sign of quotient and remainder as required.
- Optimized divider: one cycle per partial-remainder subtraction.



- Faster division: can't use parallel hardware as in multiplier, but faster dividers generate multiple quotient bits per step.
- MIPS division: HI register remainder, LO register quotient.
  - Instructions: $div\ rs,rt$ | $divu\ rs,rt$.

CUE COLUMN

SUMMARIES

1. Operations on integers such addition and subtraction.
2. Multimedia arithmetic.
3. Division.

Chapter7. Floating Point Arithmetic

2019年3月31日 20:45

NOTE TAKING AREA

Floating point representation

- Can have very small and large numbers, such as scientific notation:

-2.34×10^{56} ← normalized

$+0.002 \times 10^{-4}$ ← not normalized

$+987.02 \times 10^{9}$ ← not normalized

  - The basic binary representation of floating point:

$\pm 1.xxxxxxx_2 \times 2^{yyyy}$

- IEEE floating-point format:

$$\pm 1.xxxxxxx_2 \times 2^{yyyy}$$

single: 8 bits
double: 11 bits

single: 23 bits
double: 52 bits

S	Exponent (yyyy+Bias)	Fraction (xxxx)

$$x = (-1)^S \times (1 + Fraction) \times 2^{(Exponent - Bias)}$$

  - The first S is **sign bit**, 0 non-negative and 1 negative.
  - Normalize significant: 1.0 <= |significant| <= 2.0, be aware that:
    - There always has a leading pre-binary-point 1 bit, so **no need to represent it explicitly (hidden bit)**.
    - Significant is fraction with "1" in the head and restored.
  - Exponent: excess representation, actual exponent + bias:
    - Ensure exponent is **unsigned**.
    - Single bias = 127, double bias = 1023, which is $2^{k-1}-1$, k is the width of exponent.
- Smallest and largest range of floating point:
  - For single precision: ±1.2E10-38 to ±3.4E10+38.
  - For double precision: ±2.2E10-308 to ±1.8E10+308.
- Floating-point precision:
  - Relative precision:
    - All fraction bits are **significant**.
    - ΔA/|A|=single: 2^{-23}, double: 2^{-52}.
    - Precision: single = 23 * $\log_{10}2$ ≈ 23 * 0.3 ≈ 6, double ≈ 16.

Floating point addition and multiplication

- The basic steps to do floating-point addition:

$9.999 \times 10^{1} + 1.610 \times 10^{-1}$

  - Align decimal points: shift number with smaller exponent:

$9.999 \times 10^{1} + 0.016 \times 10^{1}$

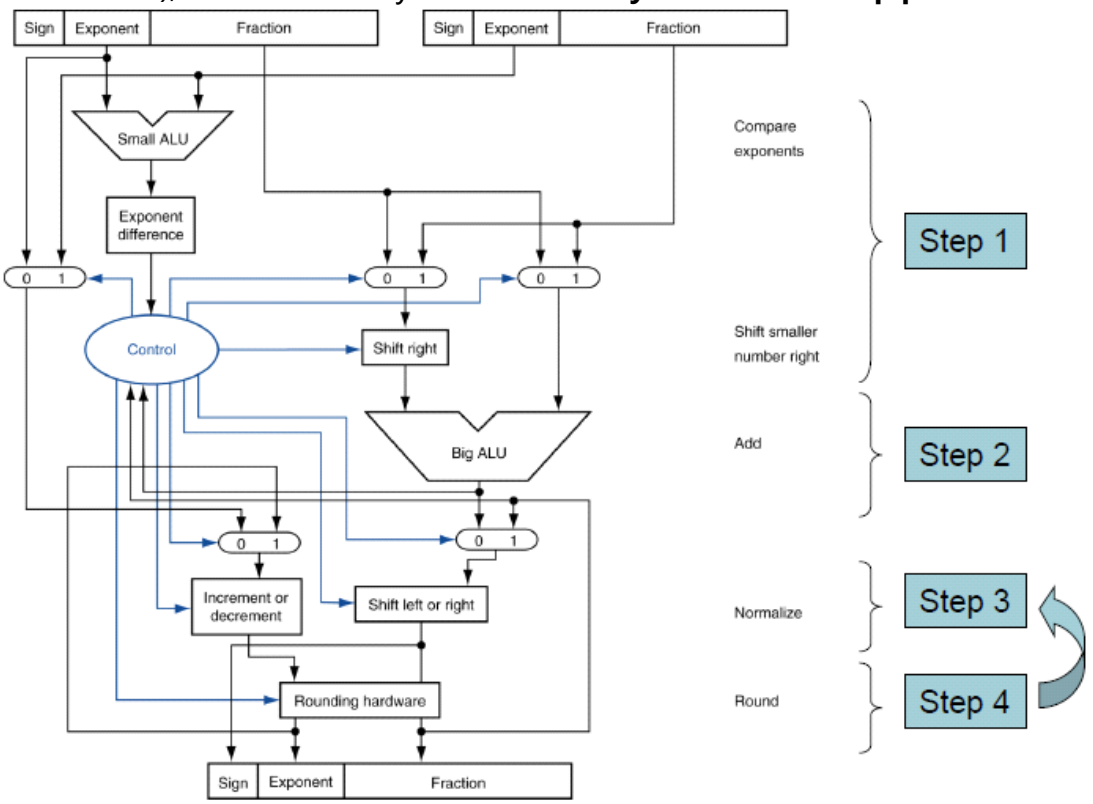
  - Add significands:

$$9.999 \times 10^1 + 0.016 \times 10^1 = 10.015 \times 10^1$$

  - Normalize result & check for over/underflow:

$$1.0015 \times 10^2$$

  - Round and renormalize if necessary.
- FP adder hardware: complex and doing in one clock cycle would take too long (much longer than integer operations, slower clock would penalize all instructions), FP adder usually **takes several cycles** and **can be pipelined**.



- FP multiplication: **compute exponent**, **multiply significands** (set the binary point correctly), **normalize**, **round** (potentially re-normalize), and **assign sign**.
- FP arithmetic hardware: addition, subtraction, multiplication, division, reciprocal, and square-root.
  - Operations usually **takes several cycles** and **can be pipelined**.

Floating point MIPS instructions

- FP hardware is coprocessor 1, which is adjunct processor that extends the ISA.
- Separate FP registers: $f0 to $f31 32 bit registers, and can be **paired for double-precision**: $f0/$f1, $f2/$f3, …
  - FP instructions operate only on FP registers, more registers with minimal code-size impact.
- FP load and store instructions: $lwc1, ldc1, swc1, sdc1$.
- Single-precision arithmetic: $add.s, sub.s, mul.s, div.s$.
- Double-precision arithmetic: $add.d, sub.d, mul.d, div.d$.
- Single and double precision comparison: $c.xx.s, c.xx.d(xx\ can\ be\ eq, lt, le, \dots)$.
  - This comparison will set or clear FP condition-code bit.
- Branch on FP condition code true or false: $bc1t, bc1f$.
  - If FP condition-code is true/false then jump to label.
- Accurate arithmetic: IEEE std 754 specifies additional rounding control: extra bits of precision(guard, round, sticky), choice of rounding modes, and allows programmer to fine-tune numerical behavior of a computation.
  - Trade-off between hardware complexity, performance, and market requirements.

Sub word parallelism

- Performing simultaneous operations on short time: graphics and audio applications.
  - Example: 128-bit adder can be represent as 16 8-bit adds, 8 16-bit adds, and 4 32-bit adds.
  - *Also called **data-level parallelism**, vector parallelism, or single instruction, multiple data (SIMD).*
- Streaming SIMD Extension 2 (SSE2): adds 4 * 128-bit registers extended to 8 registers in AMD64/EM64T.
- Optimize matrix multiply:
  - The optimized C code:

```
1. #include <x86intrin.h>
2. void dgemm (int n, double* A, double* B, double* C)
3. {
4.  for ( int i = 0; i < n; i+=4 )
5.   for ( int j = 0; j < n; j++ ) {
6.    __m256d c0 = _mm256_load_pd(C+i+j*n); /* c0 = C[i][j]
   */
7.    for( int k = 0; k < n; k++ )
8.     c0 = _mm256_add_pd(c0, /* c0 += A[i][k]*B[k][j] */
9.               _mm256_mul_pd(_mm256_load_pd(A+i+k*n),
10.              _mm256_broadcast_sd(B+k+j*n)));
11.   _mm256_store_pd(C+i+j*n, c0); /* C[i][j] = c0 */
12.  }
13. }
```

  - Optimized x86 assembly code:

```
1. vmovapd (%r11),%ymm0       # Load 4 elements of C into %ymm0
2. mov %rbx,%rcx              # register %rcx = %rbx
3. xor %eax,%eax              # register %eax = 0
4. vbroadcastsd (%rax,%r8,1),%ymm1 # Make 4 copies of B element
5. add $0x8,%rax              # register %rax = %rax + 8
6. vmulpd (%rcx),%ymm1,%ymm1 # Parallel mul %ymm1,4 A elements
7. add %r9,%rcx               # register %rcx = %rcx + %r9
8. cmp %r10,%rax              # compare %r10 to %rax
9. vaddpd %ymm1,%ymm0,%ymm0   # Parallel add %ymm1, %ymm0
10. jne 50 <dgemm+0x50>       # jump if not %r10 != %rax
11. add $0x1,%esi             # register % esi = % esi + 1
12. vmovapd %ymm0,(%r11)      # Store %ymm0 into 4 C elements
```

CUE COLUMN

Floating point example

Represent –0.75

- $-0.75 = (-1)^1 \times 1.1_2 \times 2^{-1}$
- S = 1
- Fraction = $1000...00_2$
- Exponent = –1 + Bias
  - Single: –1 + 127 = 126 = 01111110_2
  - Double: –1 + 1023 = 1022 = 01111111110_2

Single: 10111111010000...00

Double: 10111111111010000...00

Example for floating point addition

Now consider a 4-digit binary example

- $1.000_2 \times 2^{-1} + -1.110_2 \times 2^{-2}$ (0.5 + –0.4375)

1. Align binary points

- Shift number with smaller exponent
- $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1}$

2. Add significands

- $1.000_2 \times 2^{-1} + -0.111_2 \times 2^{-1} = 0.001_2 \times 2^{-1}$

3. Normalize result & check for over/underflow

- $1.000_2 \times 2^{-4}$, with no over/underflow

4. Round and renormalize if necessary

- $1.000_2 \times 2^{-4}$ (no change) = 0.0625

FP example from °F to °C

- C code of program:

```
float f2c (float fahr) {
  return ((5.0/9.0)*(fahr - 32.0));
}
```

  - Variable fahr in $f12, result in $f0, literals in global memory space.
- Compiled MIPS code:

```
f2c: lwc1  $f16, const5($gp)
     lwc1  $f18, const9($gp)
     div.s $f16, $f16, $f18
     lwc1  $f18, const32($gp)
     sub.s $f18, $f12, $f18
     mul.s $f0,  $f16, $f18
     jr    $ra
```

FP example: array multiplication

- X = X + Y * Z: all 32 by 32 matrices, 64-bit double-precision elements.
- C code of program:

```
void mm (double x[][],
         double y[][], double z[][]) {
  int i, j, k;
  for (i = 0; i! = 32; i = i + 1)
    for (j = 0; j! = 32; j = j + 1)
      for (k = 0; k! = 32; k = k + 1)
        x[i][j] = x[i][j]
                  + y[i][k] * z[k][j];
}
```

  - Addresses of x, y, z in $a0, $a1, $a2, and i, j, k in $s0, $s1, $s2.

- MIPS code:

```
    li   $t1, 32       # $t1 = 32 (row size/loop end)
    li   $s0, 0        # i = 0; initialize 1st for loop
L1: li   $s1, 0        # j = 0; restart 2nd for loop
L2: li   $s2, 0        # k = 0; restart 3rd for loop
    sll  $t2, $s0, 5   # $t2 = i * 32 (size of row of x)
    addu $t2, $t2, $s1 # $t2 = i * size(row) + j
    sll  $t2, $t2, 3   # $t2 = byte offset of [i][j]
    addu $t2, $a0, $t2 # $t2 = byte address of x[i][j]
    l.d  $f4, 0($t2)   # $f4 = 8 bytes of x[i][j]
L3: sll  $t0, $s2, 5   # $t0 = k * 32 (size of row of z)
    addu $t0, $t0, $s1 # $t0 = k * size(row) + j
    sll  $t0, $t0, 3   # $t0 = byte offset of [k][j]
    addu $t0, $a2, $t0 # $t0 = byte address of z[k][j]
    l.d  $f16, 0($t0)  # $f16 = 8 bytes of z[k][j]
```

```
    sll  $t0, $s0, 5       # $t0 = i*32 (size of row of y)
    addu  $t0, $t0, $s2    # $t0 = i*size(row) + k
    sll   $t0, $t0, 3      # $t0 = byte offset of [i][k]
    addu  $t0, $a1, $t0    # $t0 = byte address of y[i][k]
    l.d   $f18, 0($t0)     # $f18 = 8 bytes of y[i][k]
    mul.d $f16, $f18, $f16 # $f16 = y[i][k] * z[k][j]
    add.d $f4, $f4, $f16   # f4=x[i][j] + y[i][k]*z[k][j]
    addiu $s2, $s2, 1      # $k k + 1
    bne   $s2, $t1, L3     # if (k != 32) go to L3
    s.d   $f4, 0($t2)      # x[i][j] = $f4
    addiu $s1, $s1, 1      # $j = j + 1
    bne   $s1, $t1, L2     # if (j != 32) go to L2
    addiu $s0, $s0, 1      # $i = i + 1
    bne   $s0, $t1, L1     # if (i != 32) go to L1
```

SUMMARIES

1. Floating point representation.
2. Floating point addition and multiplication.
3. MIPS floating point instructions.
4. Sub word parallelism.

Concluding remarks

- Bits have no inherent meaning.
  - Interpretation depends on the instructions applied.
- Computer representations of numbers.
  - Finite range and precision.
- ISAs support arithmetic.
  - Signed and unsigned integers.
  - Floating-point approximation to reals.
- Bounded range and precision.
  - Operations can overflow and underflow.

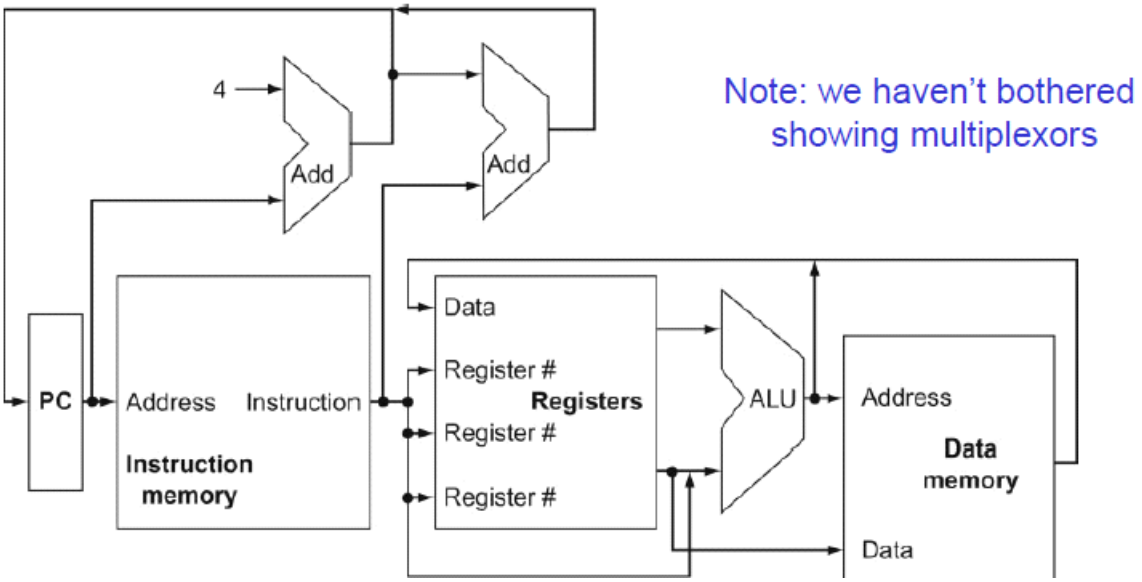
Chapter8. The Processor

2019年4月8日 14:34

NOTE TAKING AREA

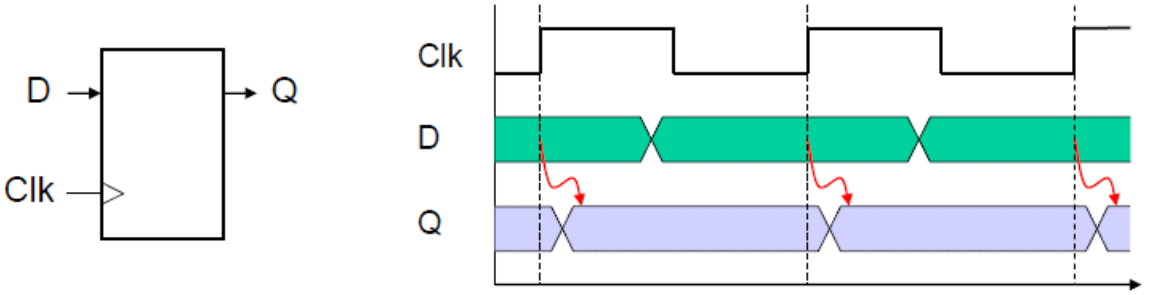
Implementation overview

- CPU time = instruction count * CPI * clock cycle time.
  - Instruction count determined by ISA and compiler.
  - CPI and cycle time determined by CPU hardware.
- Design CPU architecture:
  - Basic math, memory access, branch and jump instructions.
  - Basic parts of CPU: memory (store instructions, store data in separate units), registers, ALU, and control logic, operations to all instructions (PC and read register values).
- An example CPU:



Logic design basics

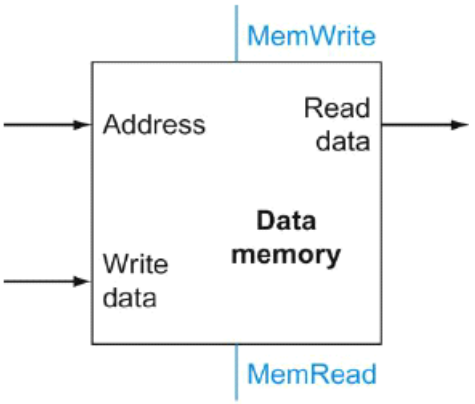
- Information encoded in binary: low and high voltage, one wire per bit, multi-bit data encoded on multi-wire buses.
- Combinational element: operate on data, output is a function of input.
  - And gate, adder, multiplexer, and arithmetic / logic unit.
- State (sequential) elements: store information.
  - Has a pre-stored state, and has some internal storage.
  - At least two inputs and one output. Data, clock, and output.



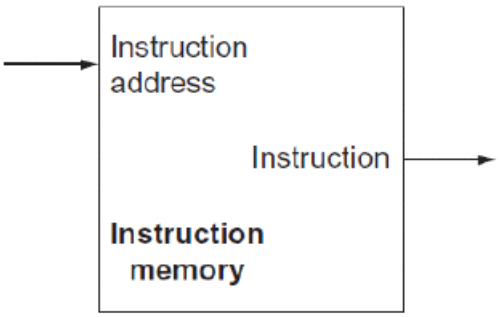
  - Sequential elements: register without write control (PC), update along with clock signal (positive edge). Register with write control (data memory / register), updates only on edge when write control input is 1, used when stored value is required later.

Memory design, registers, and clock

- Data memory:

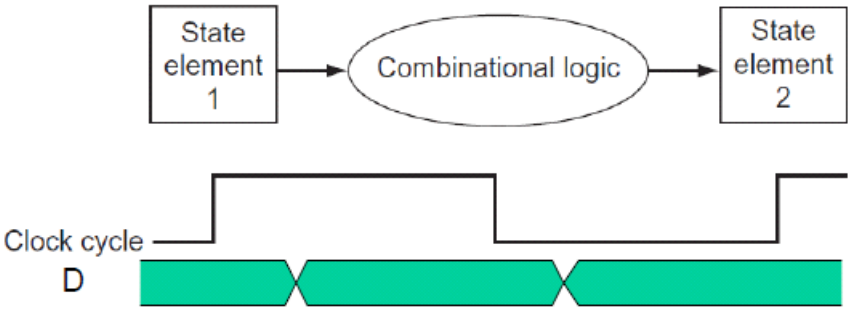


  - Input: address (32 bits), write-in data (32 bits), mem-write (1 bit), mem read (1 bit), clock.
  - Output: read-out data (32 bit).
  - State: the data stored in the memory (n * 32 bits).
- Instruction memory:



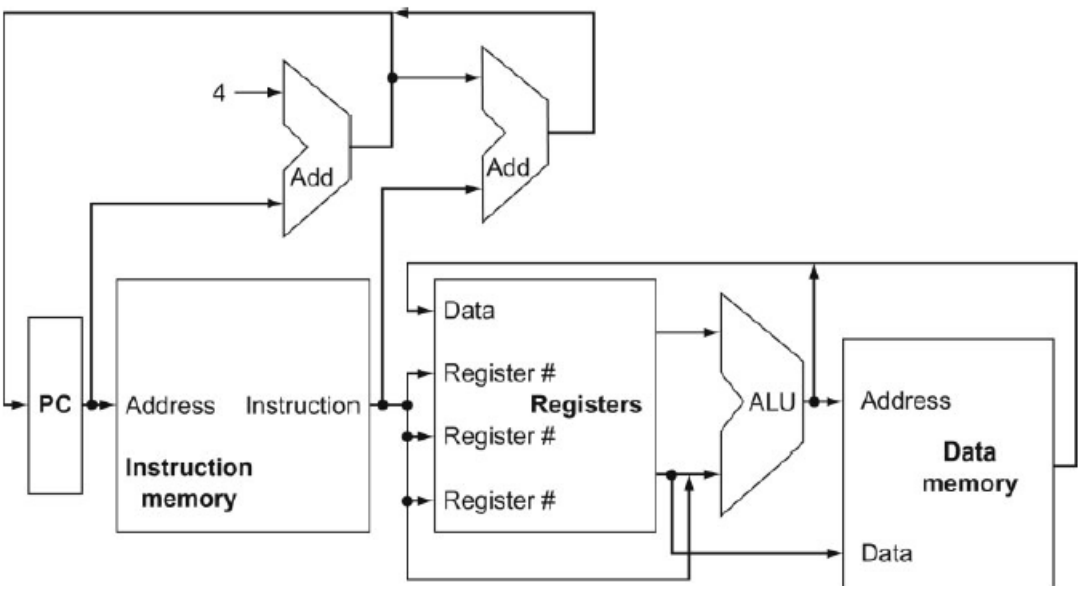
  - Input: instruction address (32 bits), clock.
  - Output: instructions (32 bits).
  - State: the instructions stored in the memory (n * 32 bits).
    - No write operation and so no control signals.
- Registers:

  - Input: three register number (5 bits * 3), write-in data (32 bits), reg-write (1 bit).
  - Output: two read-data (32 bits).
  - State: 32 * 32 bits data.
- Clock methodology: signals read and write along edges.



- Edge-triggered clocking: all state changes occur on a clock edge.
- Clock time: signals to propagate from SE1 through combinatorial element to SE2

- Clocking methodology:

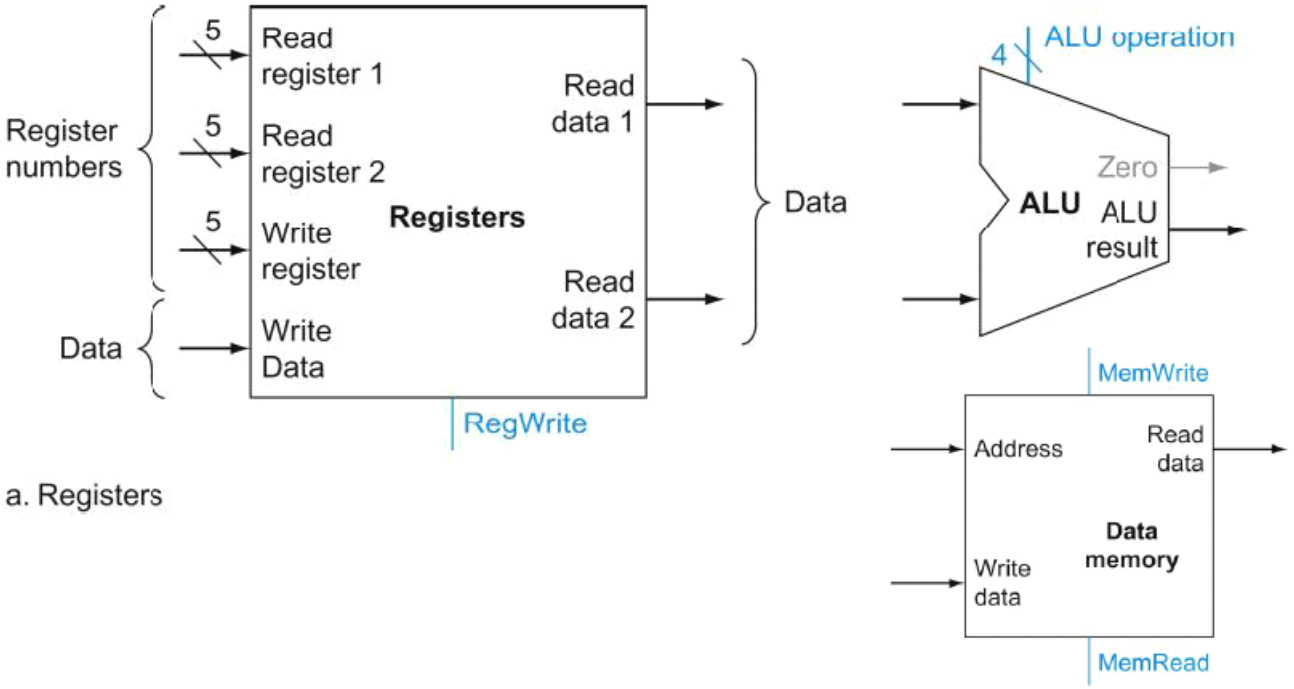


Detailed implementation for every instruction

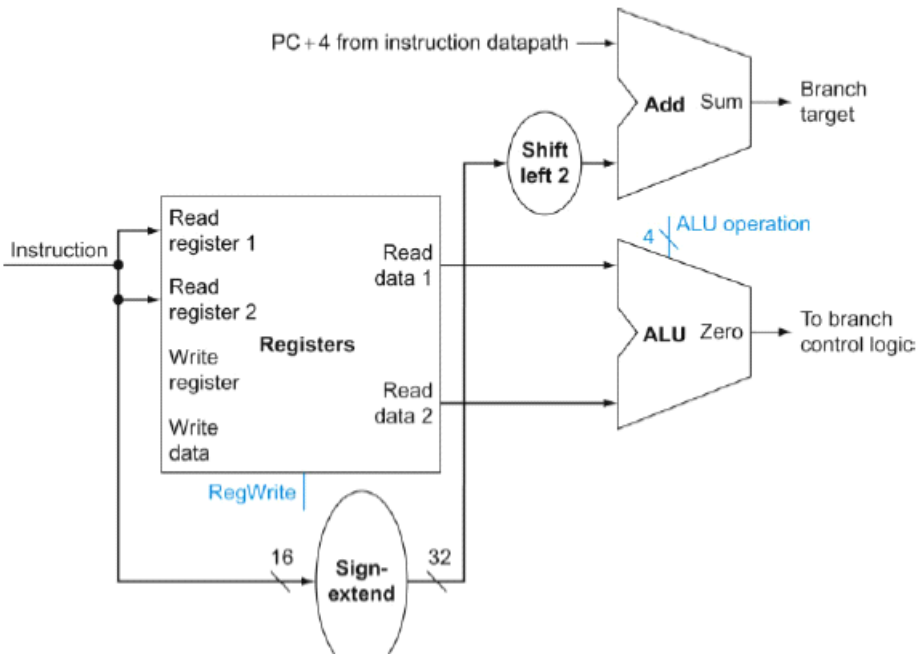
- Implementing R-type instructions:

a. Registers b. ALU

- Implementing loads / stores:



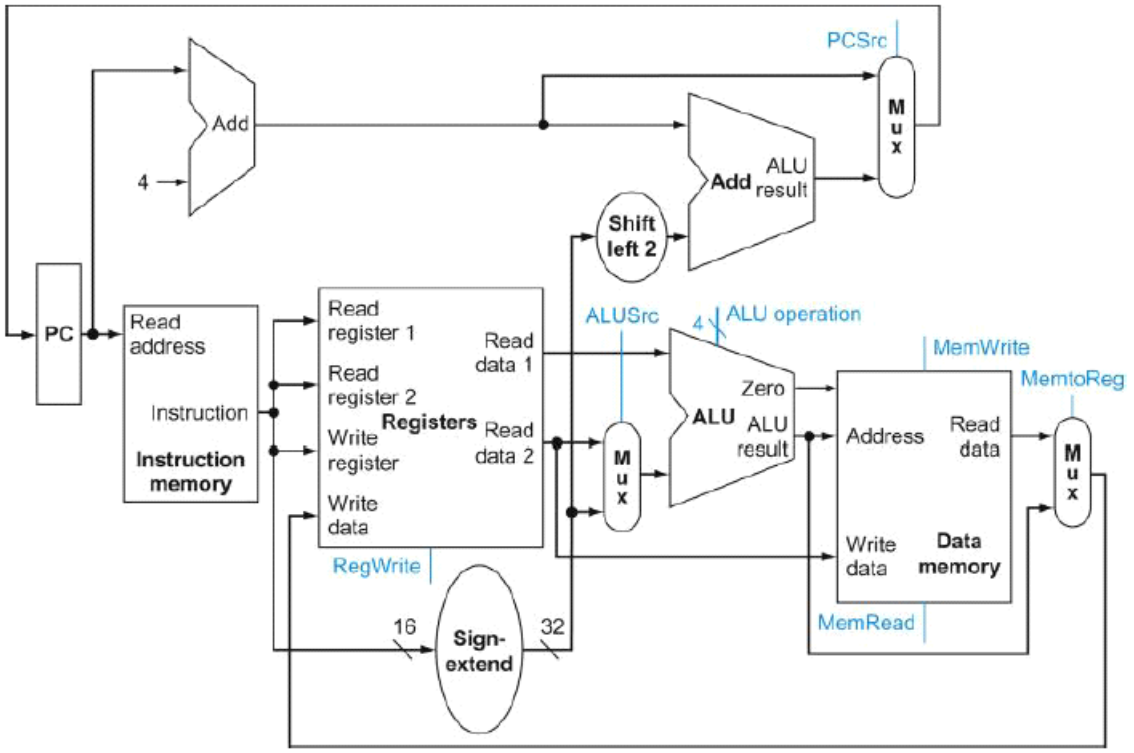
- Implementing J-type instructions:



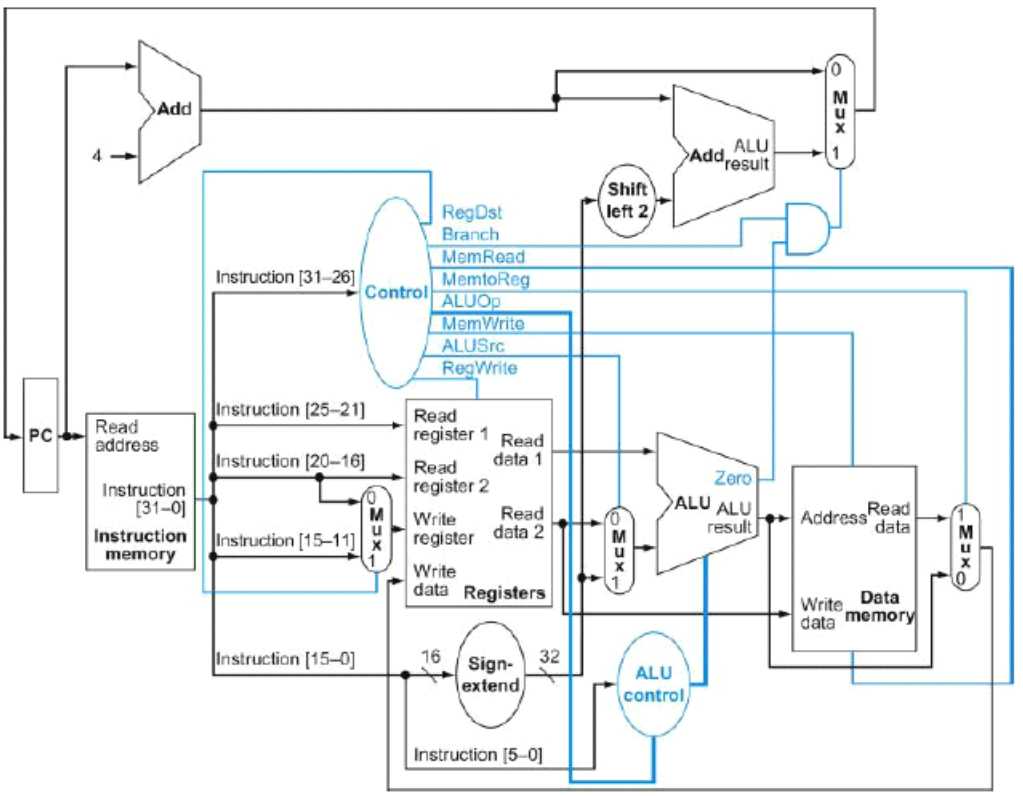
CUE COLUMN

View from high feet

- View from 10000 feet:



- View from 5000 feet:



SUMMARIES

1. Implementation overview.
2. Logic design basics.
3. Memory design, registers, and clock.
4. Detailed implementation for every instruction.

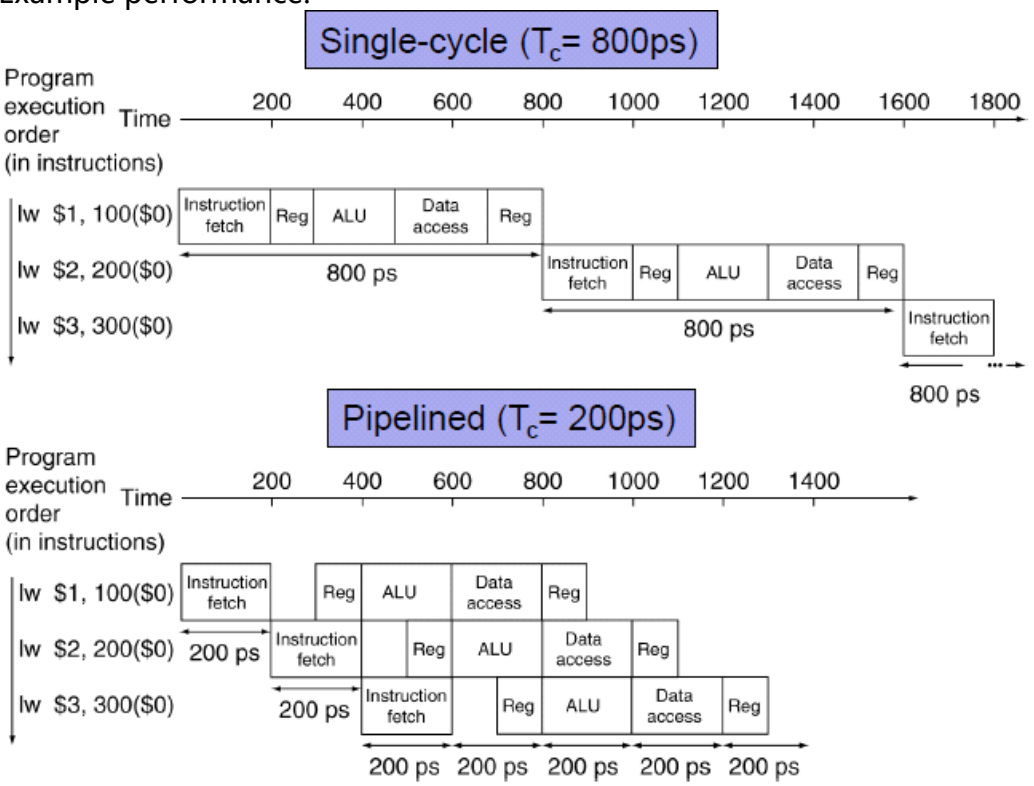
Chapter9. The Overview of Pipeline

2019年4月22日 14:13

NOTE TAKING AREA

Overview

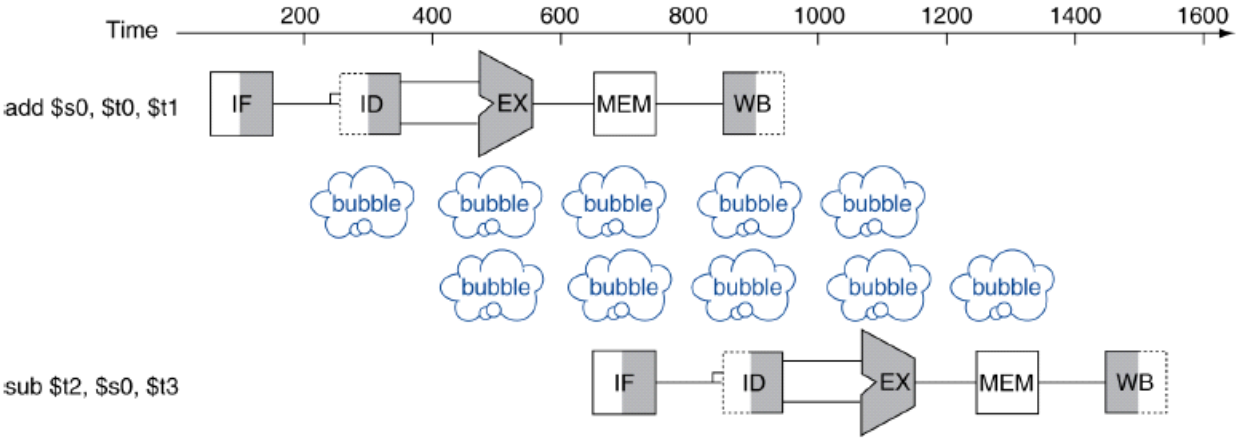
- Pipeline: used to improve performance, process multiple instructions on same time (overlapped).
- **Stages of pipeline**: IF (instruction fetch), ID (instruction decode & register read), EX (execute), MEM (access memory), WB (write to register).
  - IF: read instruction from memory.
  - ID: decode instruction, and read value from register.
  - EX: perform execution and computation.
  - MEM: find memory address to read or put value, and get value if read is needed.
  - WB: write value back to register.
- Example performance:



  - Time between instructions$_{pipilineed}$=time between instructions$_{nonpipelined}$/number of stages.
  - Pipeline may increase the time cost (latency) of single instruction.
    - Each instruction has the same latency.

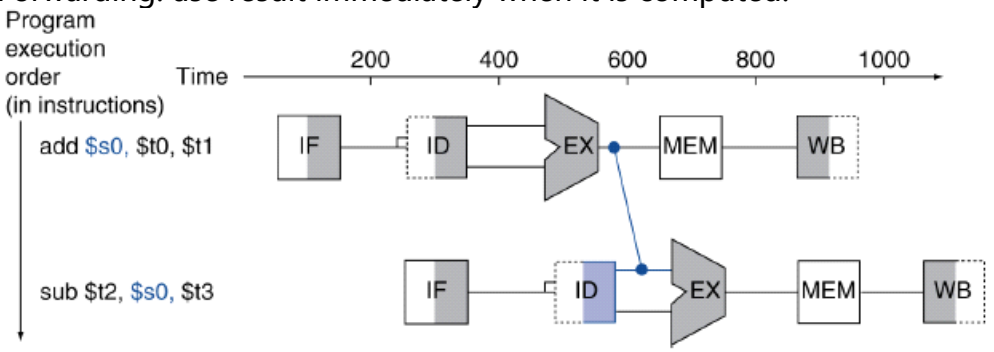
Analysis of pipeline

- ISA optimize the performance of instruction decode and memory access.
- Hazards: structure, data, control.
  - Structure hazard: memory access - separate instruction and data memory.
    - Because of bad hardware design.
  - Data hazard: data is required in next instruction.

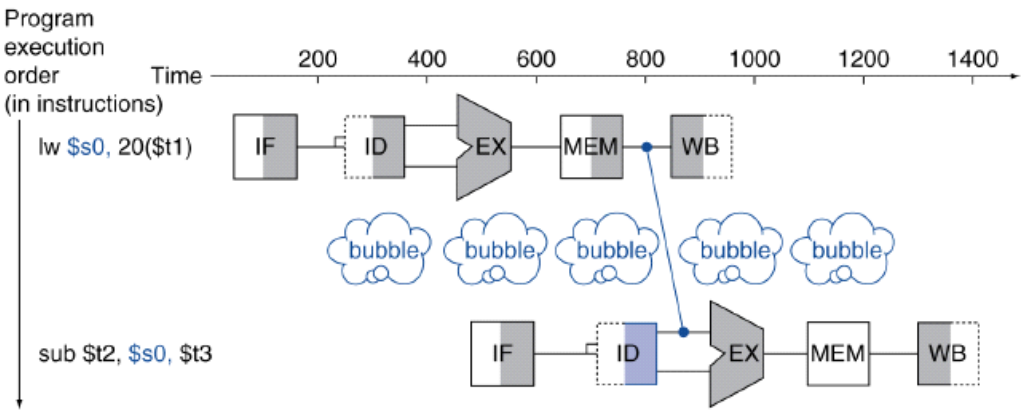


Left half black means write, and write half means read, **read and write can process in one cycle**.

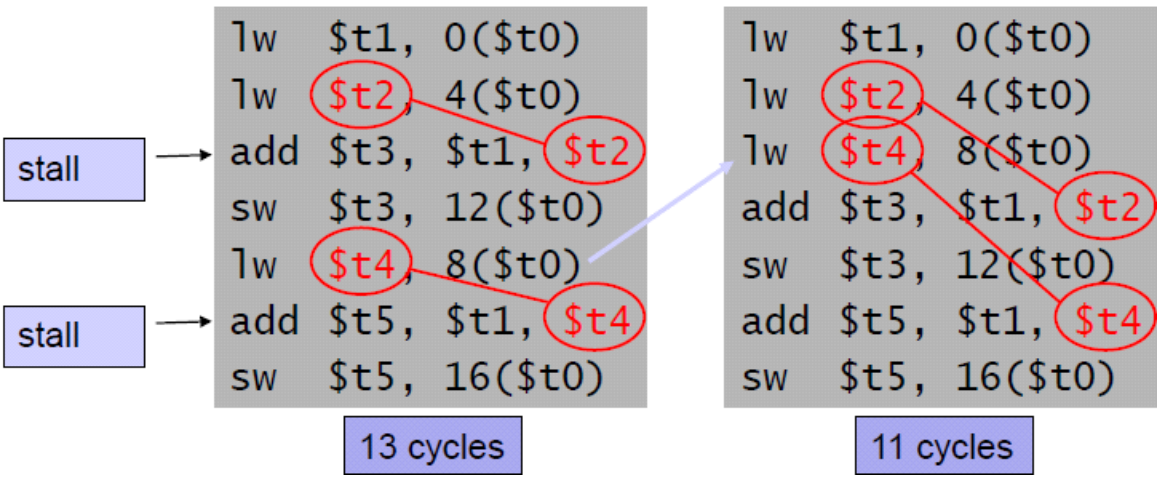
- Forwarding: use result immediately when it is computed.



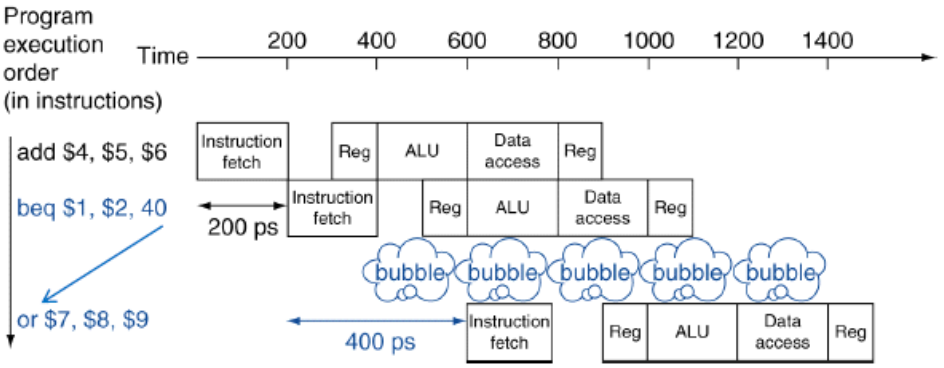
- Load-use data hazard: cannot be solved by forwarding (stall).



  - Reorder code to avoid stalls:



- Control hazard: branch determines flow of control.

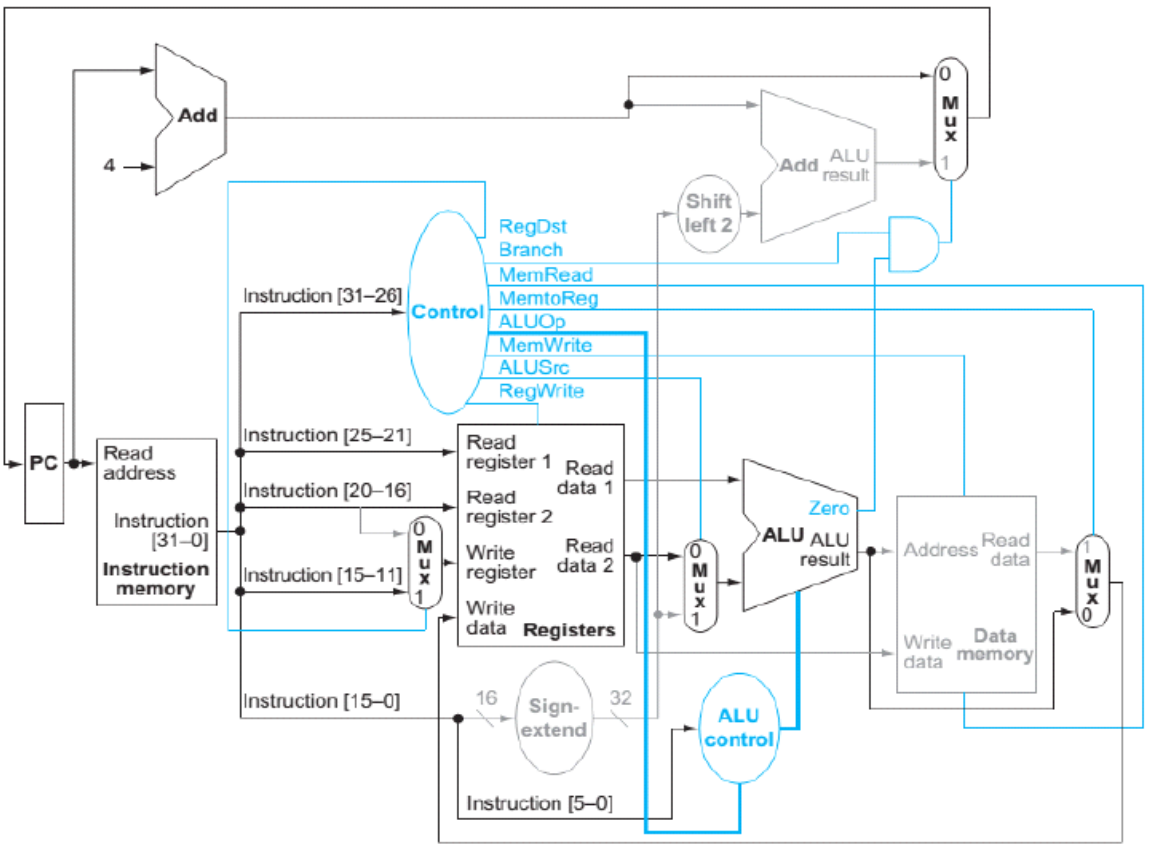


- Branch prediction: predict outcome of branch and stall only if prediction is wrong.
- More-realistic branch prediction:
  - Static branch prediction: based on typical branch behavior, go backward taken or forward not taken.
  - Dynamic branch prediction: hardware measures actual branch behavior, record recent history of each branch, and assume future behavior will continue the trend.

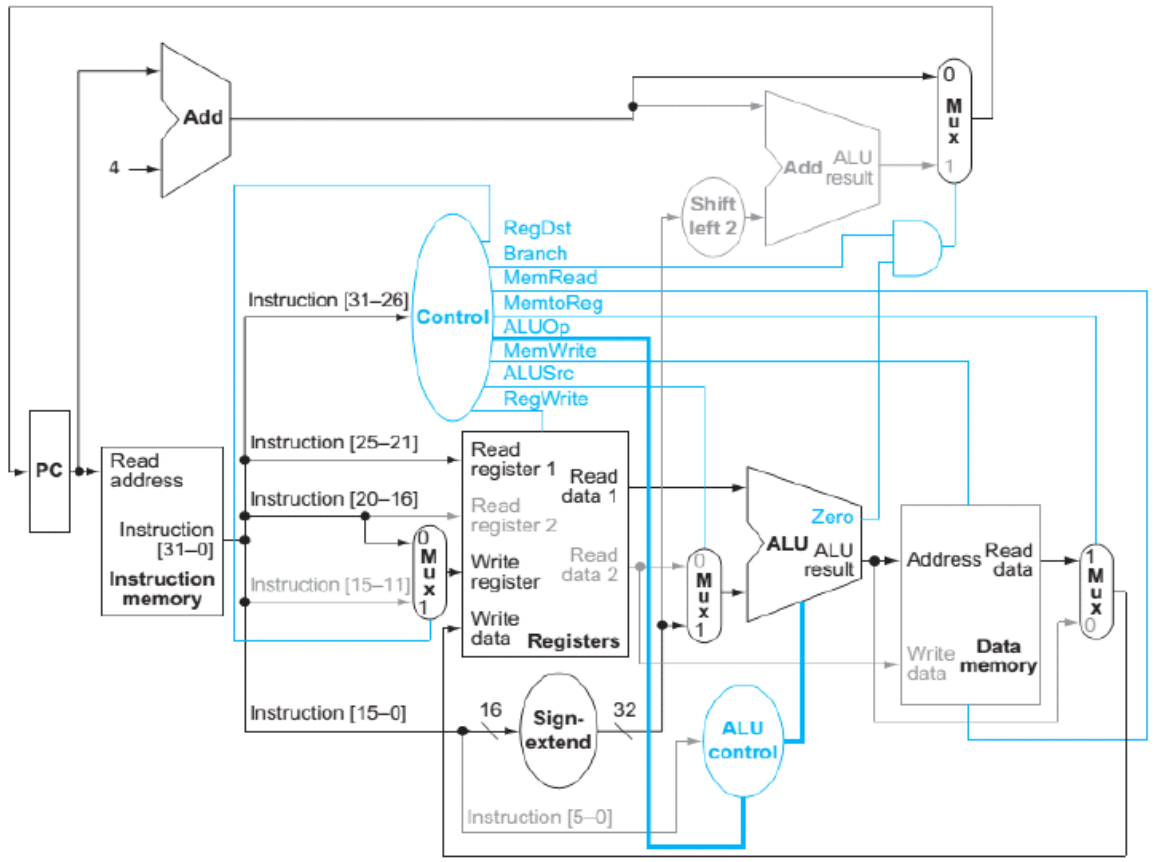
CUE COLUMN

Datapath for pipeline

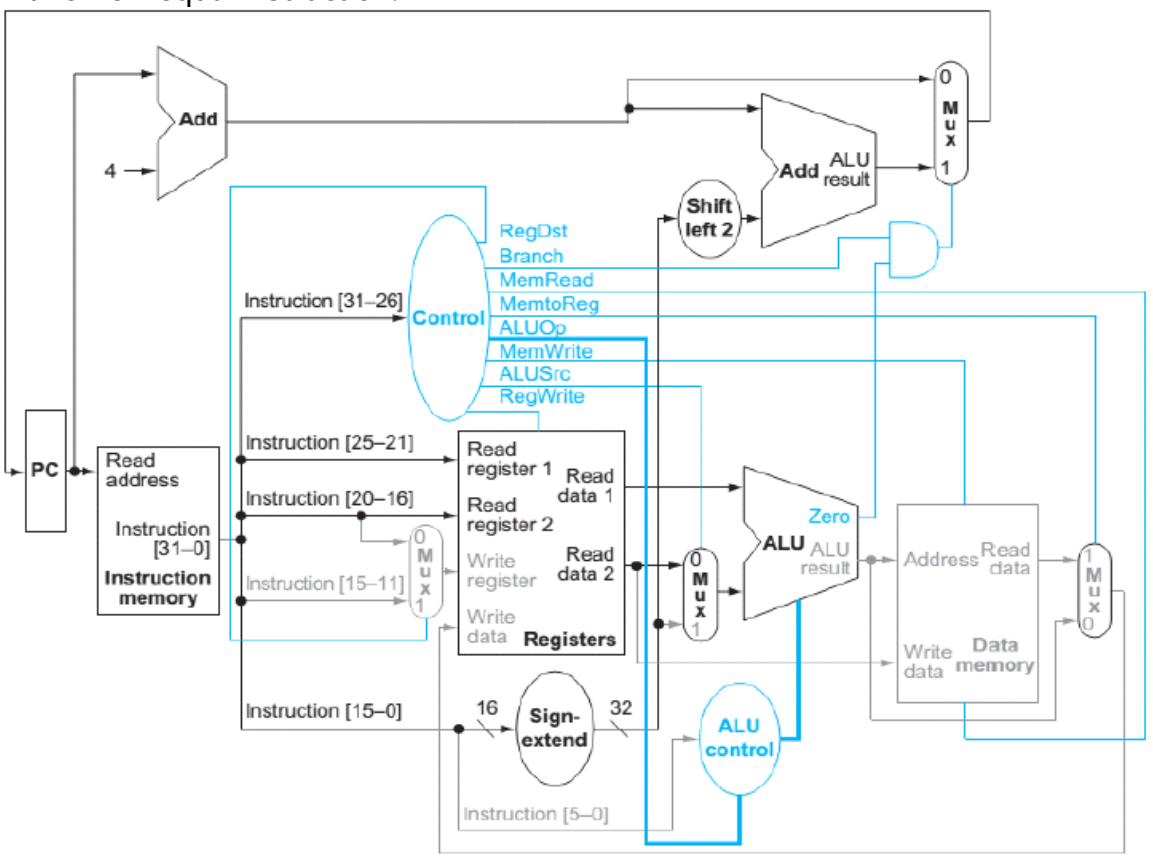
- R-type instruction:



- Load instruction:

- 

- Branch-on-equal instruction:



SUMMARIES

1. Pipelining improves performance by increasing instruction throughput.
2. Hazards of pipeline.

Chapter10. Instruction-Level Parallelism

2019年4月29日 14:16

NOTE TAKING AREA

Instruction-Level parallelism (ILP)

- Definition: parallelism among instructions. Executes multiple instructions at same time.
- IPC (instructions per second) = 1 / CPI.
- Multiple-issue: same (part of) instruction goes into CPU at same time. (issue slots)
- Static multiple issue by **compiler** and dynamic multiple issue by **processor**.
    - Static multiple issue: groups instructions into **issue packets**.
        - Two-issue packets: ALU or branch, load or store. (Goes type 1 and 2 alternately)

Hazards in static multiple issue

- Hazards: EX data hazard, ALU and followed load/store splits into two packets. Load-use hazard: still one cycle but two instructions.
- Scheduling to fix hazards: reorder - remove dependencies in one packet, or add nop.
- Loop unrolling: replicate loop body to reduce loop-control overhead. Use different registers.
- Superscalar processor: CPU decides whether to issue 0, 1, 2...
- Hardware reorder and commit result to registers in order.
- Speculation: guess and check.
    - Compiler: reorder and include "fix-up" instructions to recover from incorrect guess.
    - Hardware: look ahead and buffer results.
- Exceptions: speculative load before null-pointer check.
    - Solution: static add ISA support for deferring exceptions, dynamic buffer exceptions.
- Problems: Hard eliminated dependencies, hard exposed parallelism, memory delays and limited bandwidth.

CUE COLUMN

Scheduling example

- The initial state:

```
Loop: lw   $t0, 0($s1)      # $t0=array element
      addu $t0, $t0, $s2    # add scalar in $s2
      sw   $t0, 0($s1)      # store result
      addi $s1, $s1,-4      # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:		lw $t0, 0($s1)	1
			2
			3
			4

- The finished state:

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

- IPC = 1.25, in best condition peak IPC = 2.

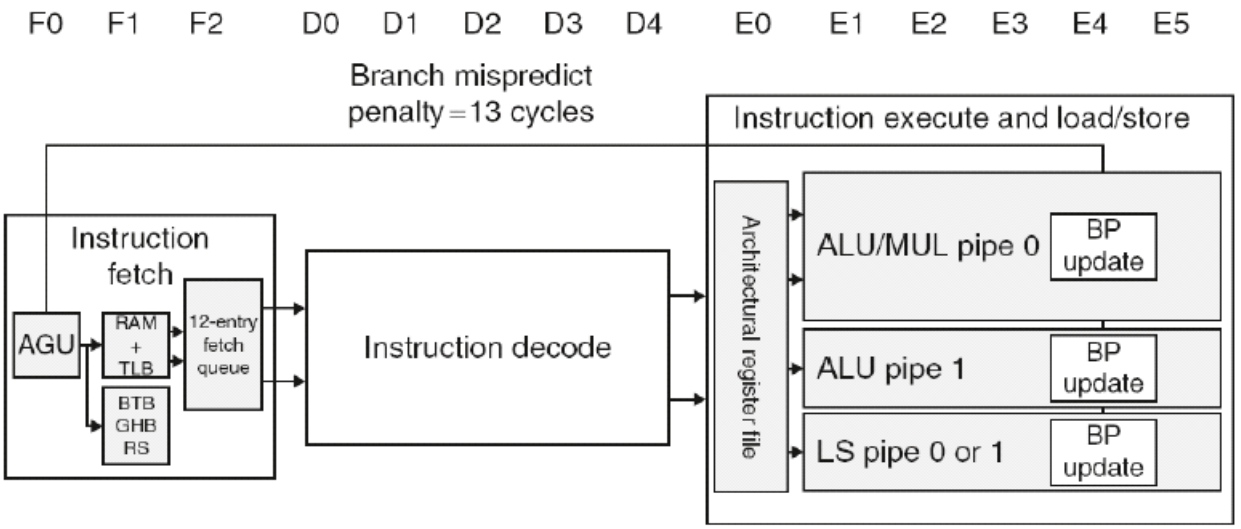
Add loop unrolling

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t3, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

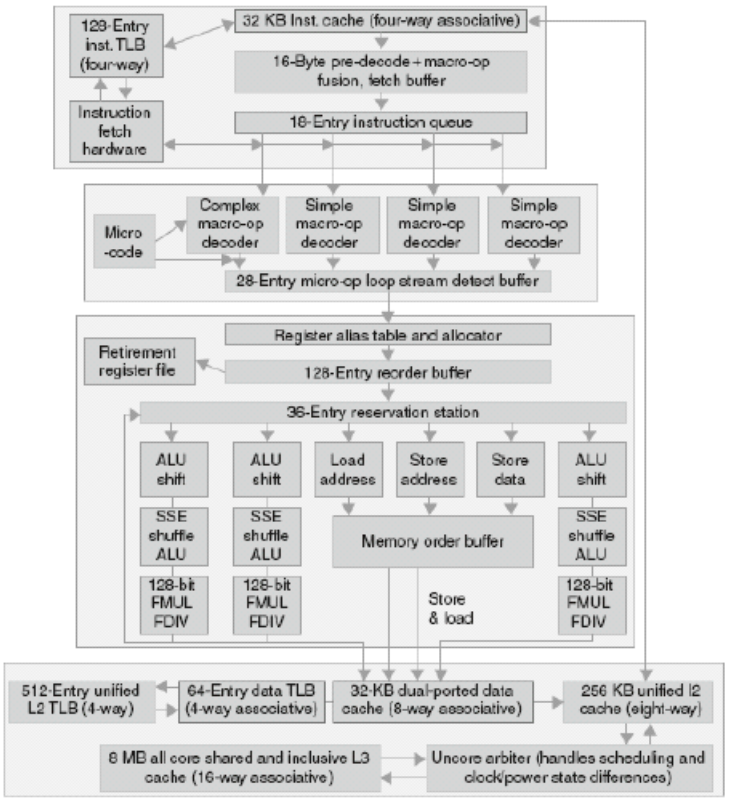
- IPC = 1.75.

Advanced overview

- Power: basic of complexity of dynamic scheduling and speculations.
- Cortex A8 and Intel i7: micro device and PC or server.
- Pipeline of Cortex A8:



- Pipeline of Core i7:



SUMMARIES

1. Instruction-level parallelism and multiple-issue.
2. Hazards and solutions.