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:

print_lnt 
p rutt_double 
print_string 
read int 
read 
double 
read_string 
read 
Char 
open 
Service 
Sys e 
print_noat 
p rilt_char 
Sao z char 
Sao = result 
_ t m call 
2 
5 
9 
10 
11 
13 
code 
Sao integer 
Sf12= float 
Sf12 = double 
Sao z SYing 
Sao = buffer. sal = 
Sao 
Sao z nename (string). Sal = flags. Sa2 mode 
Sao file descriptor. Sal buffer, Sa2 length 
Sao = file descriptor. Sal = buffer. Sa2 = length 
Sao = file descriptor 
integer (in SvO) 
float (in ST) 
double (in SfO) 
address (in $vO) 
char (in SVO) 
file descriptor (in SVO) 
num chars read (in SVC)) 
num chars written (in SvO) 

* An example assembly source file:

. text 
. globl main 
Ia Șal), str 
li ȘVO, 
sysca11 
li ȘVO, 20 
sysca11 
. data 
str: 
. ascxxz 
"he11c 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: Px / PY = 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.

4—Clock 
Clock (cycles) 
Data transfer 
and computation 
Update state 
o 
O 
O 

* 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:

n 
CPIi x Instruction Counti) 
Clock Cycles = 
i=l 

* Weighted average CPI:

Clock Cycles 
CPI = 
Instruction Count 
n 
CPII x 
Instruction Counti 
Instruction Count 

* Instruction counti / 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, TC).
* **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 \* voltage2 \* 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? 
80 
20 
+20 
n 
• Can't be done! 

* Corollary: make the common case fast.
* Fallacy: low power at idle.

**CUE COLUMN**

CPU time example

Computer A: 2GHz clock, IOS 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:

Clock CyclesB 
Clock RateB = 
CPU TimeB 
1.2 x Clock CyclesA 
6s 
Clock CyclesA 
= CPU TimeA x Clock RateA 
= IOS x 2GHz = 20 x 109 
1.2 x 20 x 109 
24 x 109 
Clock RateB = 
6s 
6s 
= 4GHz 

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.

CPU Time 
CPU Time 
CPU Time 
CPU Time 
= Instruction Count x CPIA x Cycle Time 
= Instruction Count x CPIB x Cycle Time 
I x 600ps 
..by this much 
Ix500ps 

Reducing power example

Suppose a new CPU has 
• 85% of capacitive load of old CPU 
• 15% voltage and 15% frequency reduction 
cold XO.85)2 
= 0.854 =o.52 
old 
Cold X Vold2 X Fold 

Assemble program design

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

example 
varl : 
arrayl : 
array2 : 
stringl 
. word 
.byte 
. space 
3 
40 
# create a single integer: 
#variable with initial value 3 
create a 2—e1ement character 
array with elements initialized: 
to a and b 
allocate 40 consecutive bytes, 
with storage uninitialized 
could be used as a 40—eIement 
character array, or a 
10—e1ement integer array; 
a 
.asciiz "Print this. 
comment should indicate it. 
*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 機 号 
使 用 规 思 
0 
恒 为 0 
16 
保 存 临 时 值 （ 过 程 调 用 预 留 ） 
为 汇 编 器 保 留 
口 
保 存 临 时 值 （ 过 程 调 用 预 留 ） 
2 
表 达 式 求 值 以 及 函 数 的 结 果 
保 存 时 值 （ 过 程 调 用 预 留 ） 
表 达 式 求 值 以 及 函 数 的 结 果 
四 
保 存 临 时 值 （ 过 程 调 用 預 留 ） 
保 存 临 时 值 （ 过 程 调 用 預 留 ） 
21 
保 存 临 时 值 （ 过 程 调 用 预 留 ） 
22 
保 存 临 时 值 （ 过 程 调 用 预 留 ） 
23 
保 存 临 时 值 （ 过 程 凿 用 櫝 留 冫 
8 
临 时 （ 不 为 过 程 到 用 留 〕 
时 （ 不 为 过 程 渊 用 留 ） 
临 时 （ 不 为 过 程 谲 用 预 留 ） 
2 ， 
临 时 （ 不 为 过 程 调 用 预 留 ） 
临 时 （ 不 为 过 程 调 用 预 留 ） 
为 OS 内 核 保 留 
临 时 （ 不 为 过 程 调 用 預 ） 
27 
为 内 核 保 留 
临 时 （ 不 为 过 程 调 用 楨 留 ） 
28 
全 局 区 的 指 针 
13 
时 （ 不 为 过 程 用 櫝 留 ） 
四 
堆 栈 指 针 
临 时 （ 不 为 过 程 调 用 櫝 留 ） 
临 时 （ 不 为 过 程 到 用 预 的 ） 
31 
返 回 地 址 （ 函 数 用 用 ） 
图 6 ． 1 MIPS 寄 存 器 和 使 用 能 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.

$t0, 
varl 

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

计算机生成了可选文字:
$t2, 
($t0) 

* Based or indexed addressing: using offset addressing.

Iw 
4 ($tO) 

* The front number is the offset, which can be positive or negative integers.
* Arithmetic instructions:

$tl, $t2 
add 
addi 
addu 
addu 
$t6, $t7 
subu 
subu 
$t0, 
$tl, 
$tl, 
add as 
integers 
signed 
(2' s complement) 
$t2 = $t3 D $t4 
5 
"add immediate" 
$t6,5 
sub immediate) 
(no 
$tl = 
add as unsigned integers 
$tl = $t6 
$tl = $t6 
5 
subtract as unsigned integers 

mul t 
div 
mfhi 
m f 10 
move 
$tl 
$t2,$t3 
multiply 32—bit quantities in $t3 
and $t4, and store 64—bit 
result in special registers Lo 
and Hi: 
(integer quotient) 
Hi = $t5 mod $t6 
(remainder) 
move quantity in special register Hi 
to $tO: 
$to = HI 
move quantity in special register LO 
to $tl: 
$ tl Lo, used to get at 
result of product or quotient 
$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.**
    - **Regularity** makes implementation simpler.
    - **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.

计算机生成了可选文字:
int 
Memory 
Base address 

* **Memory address is in unit of byte.**
* Registers and memory:
  1. Registers are faster to access than memory.
  2. Operating on memory data requires loads and stores.
  3. Compiler must use registers for variables as much as possible.
* Numeric representation: signed / unsigned / sign extension.
  + Unsigned binary integers: X = Xn-12n-1 + Xn-22n-2 + … + X121 + X020.
    - Range from 0 to 2n-1. For 32 bit integer, it's 0 to +4,294,967,295.
  + 2s-complement signed integer: X = -Xn-12n-1 + Xn-22n-2 + … + X121 + X020.
    - Range from -2n-1 to +2n-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

计算机生成了可选文字:
Description 
Add with Overflow 
Add without Overflow 
AND 
Divide Signed 
Divide Unsigned 
Exclusive-OR 
Multiply 
Multiply with Overflow 
Multiply with Overflow Unsigned 
NOT OR 
OR 
Set Equal 
Set Greater 
Set Greater/Equal 
Set Greater/Equal Unsigned 
Set Greater Unsigned 
Set Less 
Set Less/Equal 
Set Less/Equal Unsigned 
Set Less IJnsigned 
Set Not Equal 
Subtract with Overflow 
Subtract without Overflow 
Op-code 
add 
addu 
and 
div 
divu 
xor 
mul 
mulo 
mulou 
nor 
seq 
Sgt 
sge 
sgeu 
sgtu 
slt 
sleu 
sltu 
sne 
sub 
subu 
Operand 
destination, srcl , src2 
destination, srcl , src2 
destination, srcl immediate 
destination/srcl immediate 

* Shift operation

计算机生成了可选文字:
Description 
Rotate Left 
Rotate Right 
Shift Right Arithmetic 
Shift Left Logical 
Shift Right Logical 
Absolute Value 
Negate with Overflow 
Negate without Overflow 
NOT 
Move 
Multiply 
Multiply Unsigned 
Op-code 
rol 
srl 
abs 
neg 
negu 
not 
move 
mult 
multu 
Operand 
destination, srcl 
destination/srcl 
destination,srcl 
srcl ,src2 

* 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

* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image022.png
  + C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image023.png
* More conditional operations:
  + C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image024.png
  + C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image025.png
  + C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image026.png
* Pseudo instructions: there is **no** such instructions in hardware, the assembler **translates** them into a combination of real instructions.

• blt $sO, $sl, Label 
• If SO<SI, jump to Label 
bgt $s0, $sl, Label 
• If jump to Label 
• ble $s0, $sl, Label 
• If jump to Label 
• beqz $sO, Label 
• If jump to Label 
• Load immediate, to = 5 
• Move $to, $so 
• to-so 

* 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:

Caller: 
int x = 5 
int y = 3 
int z 
= add (x , 
Data Memory 
so 
sl 
y); 
callee: 
Register 
5 
3 
8 
int add ( int a, int b) 
intc=a + b; 
return c; 
Program Memory 
int z = add (x , y); 
int add ( int a, int b) 
{ intc=a + b; 
return c; } 

* 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 O 
$zero 
• Regs 2-3 
• Regs 4-7 
$aO-$a3 
$tO-$t7 
• Regs 8-15 • 
• Regs 16-23: $so-$s7 
• Regs 24-25: $t8-$t9 
• Reg 28 
$gp 
: $sp 
• Reg 29 
: $fp 
• Reg 30 
. Sra 
• Reg 31 
always stores the constant O 
return values of a procedure 
input arguments to a procedure 
temporaries 
variables 
more temporaries 
global pointer 
stack pointer 
frame pointer 
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.

sp 
sp 
sp 
Stack grows 
this way 
Proc C's 
Proc B's 
Proc A's 
Low address 
values 
values 
values 
High address 
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:

sgp— 1000 
1 ooo 
pc— 0040 
8000b*. 
Dynamic data 
Static data 
Text 
Reserved 

* 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:

High address 
Low address 
Saved argument 
registers (if any) 
ved return address 
Saved saved 
registers (it any) 
Local arrays and 
structures (if any) 

* **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) 
else 
f = g-h; 
bne 
add 
Else: 
Exi t: 
$s3, $s4, Else 
$so, SSI, $s2 
Exit 
sub $so, $sl, $s2 

Loop example

* Convert following code into assembly:

C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image034.png

C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image035.png

Loop: sll 
add 
bne 
addi 
Exit: 
$to, O($tl) 
$to, $s5, Exit 
$s3, $s3, 1 
Loop 

* 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 f; 
return f; 
int h, int i, int j) 
Save to,tl 
Protect environment 
The caller has saved: 
Procedure body 
g *SaC), 
h*$al, 
i*$a2, 
j*$a3, 
Restore to tl so 
return address Sra 
Return result 
leaf _ example: 
åddi-- 
$tl, 8(Ssp) 
$to, 4(Ssp) 
$so, O($sp) 
•O, -saj,-$af 
ådd 
$tl, sa2, $a3 
add 
$so, $to, sti 
sub 
SVC), $sö, $zero 
add 
$to, 4($sp) 
$tl, 8($sp) 
addi 
SSP, $sp, 12 

* 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:

(1) before: 
Data Memory 
(2) during: 
Data Memory 
so 
low address 
$sp 
high address 
$sp 
$SP+4 
$sp+8 
(3) after: 
Data Memory 
tl 
SSP 

* 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-l )); 
fact. 
addi 
sw 
g ifi 
beq 
addi 
-addi 
jal 
addi 
mul 
SSP, $sp, -8 
Sra, 4($sp) 
Sao, O($sp) 
Notes: 
The caller saves Sao 
and Sra in its stack 
space. 
Compare n<l 
Return 1 
Temps are never saved. 
Fact(n-l) 
Return ) 
$t0, $zero, Ll 
åddf 
$sp, $sp, 8 
Sra 
fact 
Sra, 4($sp) 
SSP, $sp, 8 

* 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.
* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image040.png

计算机生成了可选文字:
op 
6 bits 
rs 
5 bits 
rt 
5 bits 
immediate 
16 bits 

* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image042.png
* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image043.png

op 
6 bits 
rs 
5 bits 
rt 
5 bits 
rd 
5 bits 
5 bits 
funct 
6 bits 

* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image045.png
  + Like the register addressing, but with a shift value.
* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image046.png

计算机生成了可选文字:
op 
6 bits 
rs 
5 bits 
rt 
5 bits 
constant or address 
16 bits 

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

beq $s0,$s1, 
bne $sO,$s1, 
LI 

* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image049.png

op 
6 bits 
address 
26 bits 

Translating and starting a program

* The steps of executing a C program:

C Program x.c 
Compiler 
Assembly language program 
ssemble 
x.o 
Object: machine language module 
Linker 
x.s 
x.a, x.so 
Object: library routine (machine language) 
Executable: machine language program 
Loader 
Memory 
a.out 

* 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:

Simple portable 
Java program 
instruction set for 
the JVM 
Compiler 
Class files (Java bytecodes) Java library routines (machine language) 
In Tim 
ava Virtual Machi 
Compiles 
bytecodes of 
Compiled Java methods (machine language) 
"hot' methods 
into native 
code for host 
machine 
Interprets 
b codes 

Other popular ISA

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

**CUE COLUMN**

Target addressing example

. sll 
Loop • 
add 
bne 
addi 
j 
Exit: 
St1 , 
sti , 
Sto , 
ss3, 
Loop 
ss3, 2 
sti, ss6 
O(St1) 
Ss5, Exit 
ss3, 1 
80000 
80004 
80008 
80012 
80016 
80020 
80024 
o 
36 
5 
8 
2. 
o 
9 
9 
19 
19 
22 
8 
21 
9 
9 
20000 
4 
o 
32 

Decoding machine language table

* The table of op code:

31-29 
1(001) 
2(010) 
3(011) 
4100) 
5(101) 
6(110) 
7(111) 
R- format 
i rnrll±diete 
byte 
store by-ve 
I ink—c 
store cone _ 
11001) 
31 tz/qez 
dddiu 
hdlf 
halt 
21010) 
ump 
3(011) 
jump 8 
less 
then inliT1. 
5 tore word 
4(100) 
branch eq 
and i 
byte 
unsi 
54101) 
branch 
I Odd 
half 
unsiqed 
6(110) 
b lez 
7(111) 
botz 
immedi 

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

21010) 
(R-format), 
31011) 
0(000) 
1/001) 
2/010) 
34011) 
4/100) 
5/101) 
6/110) 
7/111) 
Ieft 
reqi',tpr 
1(001) 
mthi 
addu 
shift r tçht gra 
Int I o 
Luttract 
41100) 
sy3caII 
54101) 
Srpak 
64 110) 
xor 
7(111) 
srav 
(non) 

**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:

Carry In 
Sum = (a • • Carryln) -k (ä • b • Carryln) -l- (ä • b • Carryln) + (a • b• Carryln) 
CarryOut = (b • Carryln) + (a • Carryln) + (a • b) 
carryout 

* 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

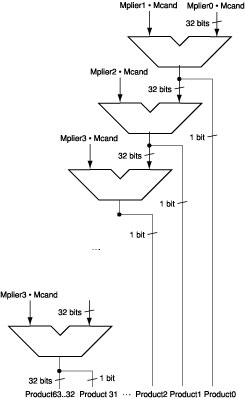
* 8 bit and 16 bit data vector will be operated on graphics and media processing.
* Use 64-bit adder with partitioned carry chain to process multimedia.
* If overflow, result will be set to the largest representable value.

multiplicand 
1000 
1000 
0000 
0000 
1000 
product 1001000 
Length of product is 
the sum of operand 
lengths 
10000000 
8 bits 
8 bits 
01001000 
8 bits 
4 bits 
0 
0 
Control test 

* 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.

Multiplicand 
32 bits 
32-bit ALU 
Shift right 
p uct 
Write 
64 bits 
Control 

* 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.
  + C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image060.png

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.

Divisor 
32 bits 
32-bit ALU 
inder 
64 bits 
Shift right 
Shift left 
Write 
Control 
test 

* 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.
  + C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image062.png

**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 x 1056 
+0.002 x 10-4 
+987.02 x 109 
normalized 
not normalized 

* The basic binary representation of floating point:

计算机生成了可选文字:
± 1.xxxxxx,K2 >< 2YYYY 

* IEEE floating-point format:

± 1.xxxxxxx2 
X 2YYYY 
S 
single: 8 bits 
double: 11 bits 
Exponent (yyyy+Bias) 
single: 23 bits 
double: 52 bits 
Fraction (xxxx) 
X = (—1 ) S X (1 + Fraction) X 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 2k-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 \* log102 ≈ 23 \* 0.3 ≈ 6, double ≈ 16.

Floating point addition and multiplication

* The basic steps to do floating-point addition:

计算机生成了可选文字:
9.999 x 101+ 1.610 x 10 1 

* Align decimal points: shift number with smaller exponent:

C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image067.png

* Add significands:

计算机生成了可选文字:
9.999 x 101 +0.016 x 101 - 10.015 x 101 

* Normalize result & check for over/underflow:

C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image069.png

* 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**.

ALO 
tep 2 
Snr 

* 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.
* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image071.png
* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image072.png
* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image073.png
* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image074.png
  + This comparison will set or clear FP condition-code bit.
* C:\265AD6E5\1020878A-10D1-4917-A5C0-0B2CB731D446.files\image075.png
  + 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. 
3. 
4. 
5. 
10. 
11. 
12. 
13. 
# include <x86intrin . 
void dgemm (int n, double* A, double* B, 
for ( int i = O; i < n; i+=4 ) 
for ( int j = 
m256d co = 
nun256 load_pd (C+i+j+n) ; 
for( int k = O; k < n; k++ ) 
double* C) 
CO — 
co +- 
mm256 mul_pd (_mm256 load_pd (A+i+k*n) 
mm256 broadcast Sd (B+k+j*n) ) ) ; 
store_pd (C+i+j co); / * C [i] = 
co 4/ 

* Optimized x86 assembly code:

2. 
3. 
4. 
5. 
6. 
10. 
12. 
vmovapd , %ymmCJ 
mov trbx, %rcx 
xor *eax, %eax 
vbroadcastsd (8rax, %r8, 1) 
add $0x8, %rax 
vrnulpd (*rcx) , eymml, eymml 
add 
c:mp Erl O, *rax 
vaddpd eyrnml, *ymmO, eymm0 
jne 50 
add $0xI, 
vmovapd %ymm0, (%rll) 
Load 4 elements 
of C into %ynunl) 
register trcx — 
% I-by. 
register *eax = O 
%ymrnl Make 4 copies of B element 
register *rax = *rax + 8 
Parallel mul eyrm1,4 A elements 
register *rcx = ercx + *1-9 
compare er10 to erax 
Parallel add *ymml, eynunO 
jump if not 
register esi — 
Store tymmO into 
! — %rax 
4 
C elements 

**CUE COLUMN**

Floating point example

Represent —O. 75 
• -0.75 = (-1)1 x 1.12 x 2-1 
• Fraction = 1000...002 
• Exponent = —1 + Bias 

Example for floating point addition

Now consider a 4-digit binary example 
1.0002 x 2-1+-1.1102 x 24 (0.5+-0.4375) 
1. Align binary points 
Shift number with smaller exponent 
1.0002 x 2-1+-0.1112 x 2-1 
2. Add significands 
1.0002 x 2-1+-0.1112 x 2-1=0.0012 >< 2-1 
3. Normalize result & check for over/underflow 
1.0002 >< 2-4, with no over/underflow 
4. Round and renormalize if necessary 
1.0002 X 24 (no change) = 0.0625 

FP example from °F to °C

* C code of program:

float f2c (float fahr) { 
return ((5 .0/9.0) 
- 32.0)); 

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

f2c: 
lwcl 
Iwcl 
div. s 
I wc I 
sub. s 
mul . s 
$f16, 
$f18, 
$f16, 
$f18, 
$f18, 
$ ra 
const5($gp) 
const9($gp) 
$f16, $f18 
const 32 ($gp) 
$f12, $f18 
$f16, $f18 

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 [ ] C] , 
double [ ] , double zC] C]) { 
int i, 
for (i = 
32; i = 1 + 1) 
for (j = 
32; j = 
for (k = 0; k! = 
32; k 

* Addresses of x, y, z in $a0, $a1, $a2, and i, j, k in $s0, $s1, $s2.
* MIPS code:

I-I: 
sll 
addu 
sll 
addu 
sll 
addu 
sll 
addu 
l.d 
Stl, 
sso, 
SSI, 
ss2, 
st2, 
st2, 
st2, 
st2, 
Sf4, 
sto, 
sto, 
sto, 
sto, 
Sf16, 
32 
sso, 
st2, 
St2 
Sao , 
5 
SSI # 
St2 
O(St2) 
ss2, 5 
sto, SSI # 
sto, 3 
sa2, Sto # 
O(StO) 
st2 = 
St2 
St2 
# Sf4 - 
Sto = 
Sto = 
Sto = 
Sto = 
s f16 
Stl = 32 (row size/ loop end) 
initialize 1st for loop 
restart 2nd for loop 
restart 3rd for loop 
* 32 (size of row of x) 
* size(row) + j 
— byte offset Of C j ] 
— byte address Of C j] 
— 8 bytes of XCi] C j ) 
k * 32 (size of row of z) 
k * size(row) + j 
byte offset Of [k] [j] 
byte address Of z[k] [j] 
= 8 bytes of z[k] [j] 

sll 
addu 
sto, ss2 
sll 
StO, 3 
addu 
l.d 
mul . d 
add. d 
addi u 
bne 
s.d 
addi u 
bne 
addi u 
bne 
sso, 
sal 
5 
, Sto 
Sto , 
Sf18, 
Sf16, 
Sf4, 
ss2, 
ss2, 
Sf4, 
SSI, 
SSI, 
sso, 
sso, 
Sto 
Sto 
Sto 
# Sf18 
= i*32 (size of row of y) 
= byte offset of [i] [k] 
= byte address of y [i) [k) 
= 8 bytes of y[i] [k] 
O(StO) 
Sf18, Sf16 # Sf16 
Sf4, Sf16 
ss2, 1 
stl, L 3 
O(St2) 
SSI, 
Stl 
sso, 
Stl 
1 
1 
if (k 
# = Sf4 
if (1 
32) go to L3 
1 
32) go to L2 
1 
32) go to Ll 

**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**

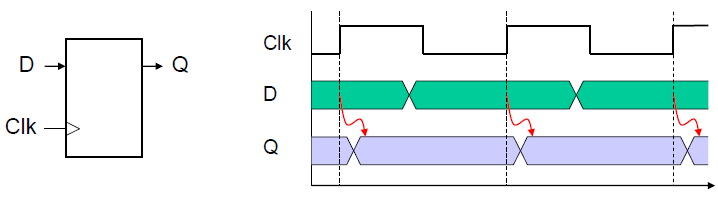
Implementation overview

* 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:

Note: we haven't bothered 
showing multiplexors 
PC 
4 
Address Instruction 
Instruction 
memory 

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:

write 
MemWrite 
data 
memory 
MemRead 

* 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:

Instruction 
address 
Instruction 
Instruction 
memory 

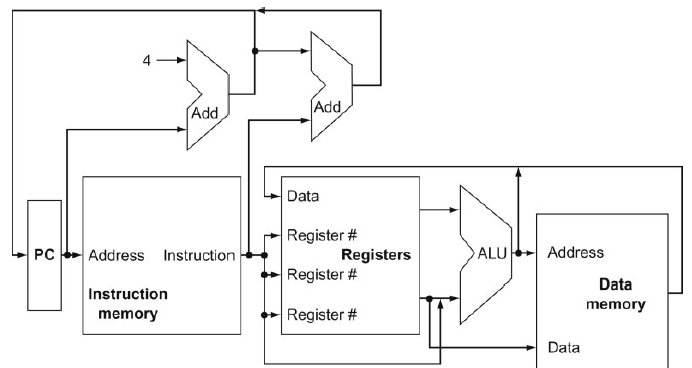
* Input: instruction address (32 bits), clock.
* Output: instructions (32 bits).
* State: the instructions stored in the memory (n \* 32 bits).
  + No write operation and so no control signals.
* Registers:

Register 
numbers 
Data 
S 
5 
5 
Read 
register 1 
Read 
register 2 
Write 
register 
Data 
Read 
data 1 
Data 
Registers 
Read 
data 2 
RegWrite 

* 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.

State 
element 
Clock cycle 
Combinational logic 
State 
element 
2 

* 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

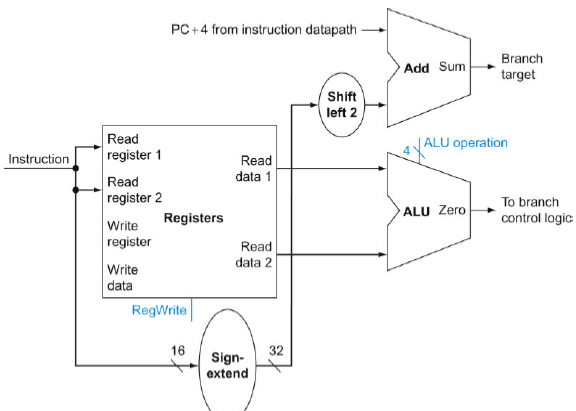
* Implementing R-type instructions:

5 
5 
5 
a. Registers 
Read 
register 1 
Read 
register 2 
Write 
register 
data 2 
RegWrite 
ALU operation 
Zero 
ALU ALU 

* Implementing loads / stores:

Register 
numbers 
Data 
a. Registers 
5 
5 
5 
Read 
register 1 
data 1 
Read 
register 2 
Registers 
Write 
register 
data 2 
Write 
RegWrite 
ALI-J operation 
4 
Zero 
ALU ALI-I 
MemW rite 
MemRead 

* Implementing J-type instructions:



**CUE COLUMN**

View from high feet

* View from 10000 feet:

ALUSrc 
re,øsW 1 
I rstruction 
ALL' operation 
Write Data 
data 

* View from 5000 feet:
* (3'-01 
  1310 
  125—2 
  ( t 5—01 
  m:.JSvc 
  Read 
  Re 
  16 
  left 2 
  ALu 

**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:

ingle-cycle (Tc 
Program 
execution 
Time 
(in instructions) 
'w $1, 100($0) 
"w $2, 200($0) 
'w $3, 300(S0) 
Pvarn 
Time 
order 
(in instructions) 
lw S', IOO(SO) 
lw S2, 200($0) 
lw SO, 300($0) 
= 800ps 
1000 1200 
200 
Reg 
200 
400 600 
800 ps 
1400 
1400 
1600 
1 BOO 
BOO ps 
800 ps 
Pi 
400 
lined 
600 
= 200ps 
Em 1000 
200 ps 
AL 
2m ps 200 ps 200ps Üps 

* Time between instructionspipilineed=time between instructionsnonpipelined/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.

1000 
MEM 
1200 
1400 
1600 
add "O, Sto, St' 
sub 92, Sso, $13 

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.

Program 
execution 
order 
Time 
(in instructions) 
Sso, $to, $11 
sub $t2, Sso, St3 
MEM 

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

Program 
execution 
(in instructions) 
Sso, 20($t1) 
sub St2, $so. St3 
1 ooo 
1200 
MEM 
1400 

* Reorder code to avoid stalls:

w 
w 
dd 
w 
Stall 
Stall 
su, 
O($tO) 
st2 
st3, $tl, st2 
st3, 
12($tO) 
Sto 
st5 
, $tl, St4 
16 ($tO 
3 
w 
w 
dd 
w 
dd 
sti, 
O($tO) 
st2 
4($tO) 
to) 
st3 , 
tl, St2 
St3, 1 to 
st5, 
$tl, 
16($tO 
es 

* Control hazard: branch determines flow of control.

Program 
execution 
Time 
order 
(in instructions) 
add S4, $5, $6 
beq Sl, $2, 40 
200 
400 
600 
ub51 
Boo 
u bbl 
1200 
1400 
200 ps 
or$7, S8, $9 
ALU 
bb 
400 ps 

* 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.

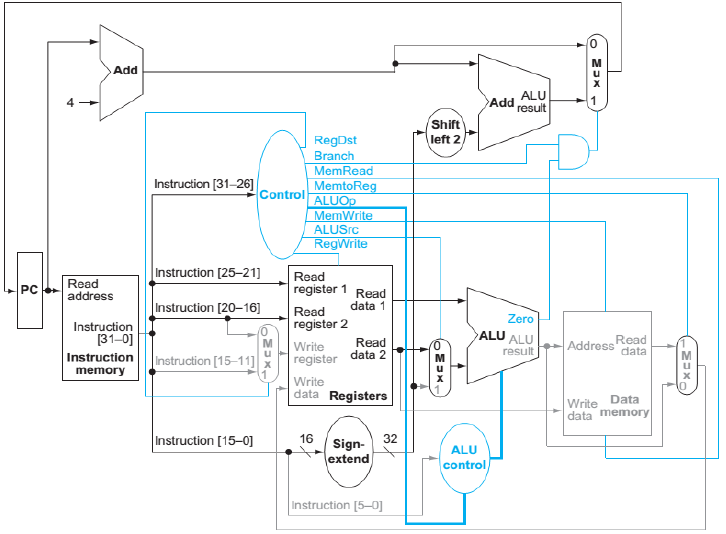
**CUE COLUMN**

Datapath for pipeline

* R-type instruction:

ernW 1 
, “ 1 23d 
ALU ALL 

* Load instruction:
* mIV 
  -21 , 00 
  0a1 一 
* 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 : 
addu 
addi 
bne 
$sl, 
$sl, 
O($sl) 
sto, $s2 
0(Ss1) 
SSI, —4 
Szero, LOOP # 
Load/store 
StO=array el ement 
add scalar in $s2 
store result 
decrement poi nter 
branch 
ALIJ/branch 
Loop: 
lw sto, O(SSI) 
cycle 
1 
2 
3 
4 

* The finished state:

ALI-J/branch 
Loop : 
nop 
addi 
addu 
bne 
SSI, 
sto, 
SSI, 
SSI, —4 
sto, ss2 
Szero, Loop 
Load/store 
nop 
nop 
sto , 
ocssl) 
4(SS1) 
cycle 
1 
2 
3 
4 

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

Add loop unrolling

ALU/branch 
-16 
Loop: 
addi 
nop 
addu 
addu 
addu 
addu 
nop 
bne 
SSI, 
sto, 
stl, 
st2, 
st3, 
SSI, 
sto, 
stl, 
st2, 
st3, 
SS2 
ss2 
SS2 
ss2 
Load/store 
sto, 
O(SSI) 
stl, 
12 (Ssl) 
st2, 
SCSI) 
st3, 
4(SS1) 
sto, 16(ss1) 
stl, 
12 (SSL) 
$t2, SCSI) 
st3, 
4(SS1) 
cycle 
1 
2 
3 
4 
6 
7 
Szero, Loop 

* 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:

计算机生成了可选文字:
Instruction 
DO DI 02 D3 
Branch mispredict 
penalty = 13 cycles 
Instruction decocle 
El 
E2 
E5 
Instruction execute and load/store 
AGU 
fetch 
12-entty 
queue 
ALCJ/MUL pipe O 
ALU pipe 1 
LS pipe O or 1 
update 
update 
update 

* Pipeline of Core i7:

计算机生成了可选文字:
I & Entry 
Ret i remsnl 
fie 
512•Entry 
ME all 
16 -gyte 
Sinvle 
Stch buffer 
Ehih 
data 
tabB 
128•Fntty reorder 
add 
Memory oor tuffet 
St o re 
32' KB dato 
28-bit 
4.81JL 
Kg 12 
and 

**SUMMARIES**

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