计算机系统结构

第四课：指令系统

王韬

wangtao@pku.edu.cn

http://ceca.pku.edu.cn/wangtao

2018

第四课：指令系统

- 数字表示方法
- 设计简单处理器
- 概念与实例

第四课：指令系统

- 数字表示方法
- 设计简单处理器
- 概念与实例

字节、字、双字

- 字（Word）的原始含义
  - 一个8位的系统，一个字原始含义是多少位？
  - 64位的系统呢？
- 通常含义
  - 字节（Byte）：8位
  - 字（Word）：16位
  - 双字（DWord）：32位

无符号数

16进制	2进制	10进制
0x00000000	0…0000	0
0x00000001	0…0001	1
0x00000002	0…0010	2
0x00000003	0…0011	3
0x00000004	0…0100	4
0x00000005	0…0101	5
0x00000006	0…0110	6
0x00000007	0…0111	7
0x00000008	0…1000	8
0x00000009	0…1001	9
	…	
0xFFFFFFFC	1…1100	2^{32} - 4
0xFFFFFFFD	1…1101	2^{32} - 3
0xFFFFFFFE	1…1110	2^{32} - 2
0xFFFFFFFF	1…1111	2^{32} - 1

2^{31} 2^{30} 2^{29} . . . 2^{3} 2^{2} 2^{1} 2^{0} **bit weight**

31 30 29 . . . 3 2 1 0 **bit position**

1 1 1 . . . 1 1 1 1 bit

1 0 0 0 . . . 0 0 0 0 – 1

2^{32} - 1 = 4,294,967,295

有符号数

- 32位补码

$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = 0_{ten}$
$0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = +\ 1_{ten}$
...

$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = +\ 2{,}147{,}483{,}646_{ten}$
$0111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = +\ 2{,}147{,}483{,}647_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000_{two} = -\ 2{,}147{,}483{,}648_{ten}$
$1000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0000\ 0001_{two} = -\ 2{,}147{,}483{,}647_{ten}$
...

$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1110_{two} = -\ 2_{ten}$
$1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111\ 1111_{two} = -\ 1_{ten}$

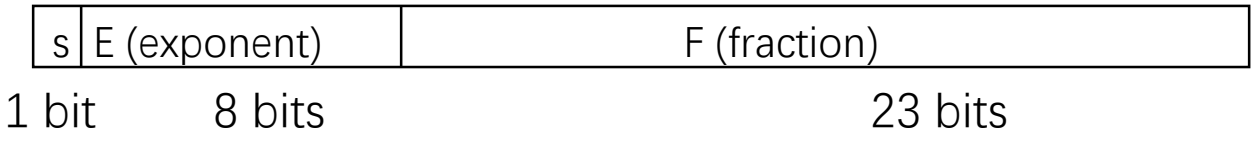
- 如何快速生成负数的二进制表示?
- 如何将非32位数扩展到32位数?
  - 0010如何扩展?
  - 1010如何扩展?

如何表示小数？

- 定点数
  - 事先确定好有多少位小数：Qx
  - Q0：没有小数，即整数
  - Q1：1位小数：0111代表多少？
  - Q2：2位小数：0111代表多少？
  - 32位有符号数，Qx中的x最大是多少？

超大和超小的数 → 浮点数

- 32位定点数的极限
  - Q0: 0 – 4,294,967,295，或者 -2,147,483,648 – 2,147,483,647
  - Q32: 非0的数最小到$1/2^{32}$ = 0.00000000023283064365…
- 如何表示地球的年龄?
  - 4,600,000,000 or 4.6×10^{9}
- 如何表示原子质量单位?
  - 0.000000000000000000000000166 or 1.6×10^{-27}
- 浮点数（课后自行参考IEEE 754）
  - $(-1)^{sign} \times F \times 2^{E}$
  - F被标准化为1.xxx…
  - 单精度32位



  - 双精度64位

字符的表示方法

- American Standard Code for Information Interchange (ASCII)

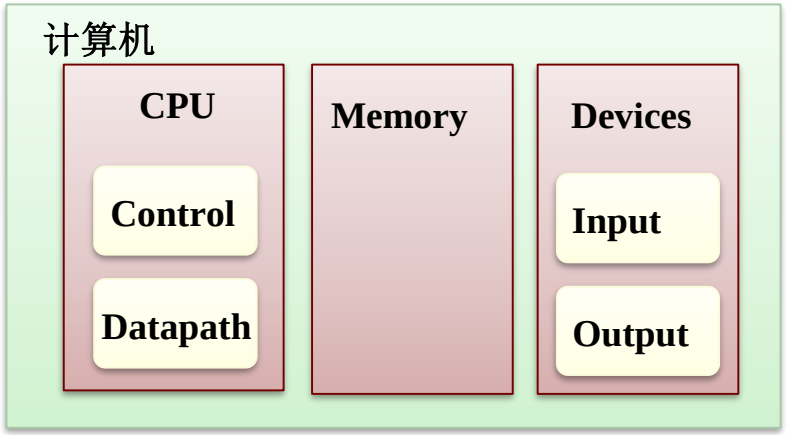
ASCII	Char		ASCII	Char	ASCII	Char	ASCII	Char		ASCII	Char	ASCII	Char
0	Null		32	space	48	0	64	@		96	`	112	p
1			33	!	49	1	65	A		97	a	113	q
2			34	“	50	2	66	B		98	b	114	r
3			35	#	51	3	67	C		99	c	115	s
4	EOT		36	$	52	4	68	D		100	d	116	t
5			37	%	53	5	69	E		101	e	117	u
6	ACK		38	&	54	6	70	F		102	f	118	v
7			39	‘	55	7	71	G		103	g	119	w
8	bksp		40	(	56	8	72	H		104	h	120	x
9	tab		41	)	57	9	73	I		105	i	121	y
10	LF		42	*	58	:	74	J		106	j	122	z
11			43	+	59	;	75	K		107	k	123	{
12	FF		44	,	60	<	76	L		108	l	124	|
15			47	/	63	?	79	O		111	o	127	DEL

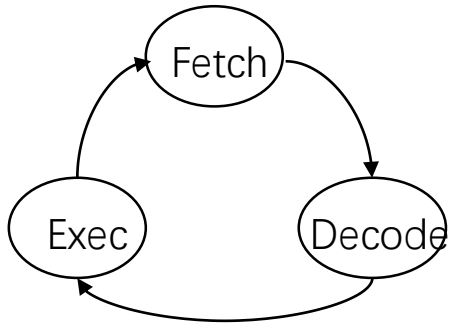
第四课：指令系统

- 数字表示方法
- 设计简单处理器
- 概念与实例

冯·诺依曼体系结构组织

- 数据通路 – Datapath，包含
  - 各功能模块
  - 执行指令所需的存储（例如寄存器）
  - 各模块之间的互联（包含支持从内存中访问数据的部分）
- 控制 - Control
  - 从内存中取指令
  - 发射信号，控制在数据通路不同模块之间的信息传递和它们的操作类型
  - 控制指令序列





复习：指令集体系结构（ISA）

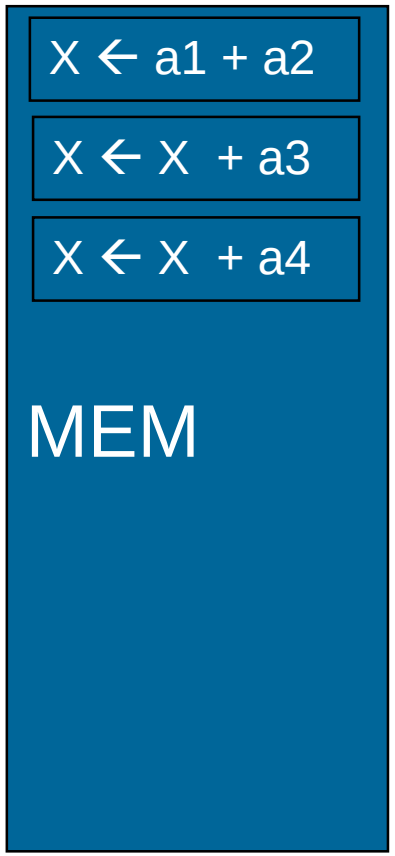
- 一台机器硬件与最底层软件之间的抽象界面，包含了编写机器语言程序所必需的所有信息，包括指令集、寄存器、存储访问方法、I/O方法等

  “... the attributes of a [computing] system as seen by the programmer, i.e., the conceptual structure and functional behavior, as distinct from the organization of the data flows and controls, the logic design, and the physical implementation.”
  – Amdahl, Blaauw, and Brooks, 1964

- 可以使相同软件运行在同一指令集体系结构的不同型号计算机/处理器上（它们具有不同的价格和性能）

简单处理器：都需要哪些指令？

$$X \leftarrow \sum_{i=1}^{N} a_i$$



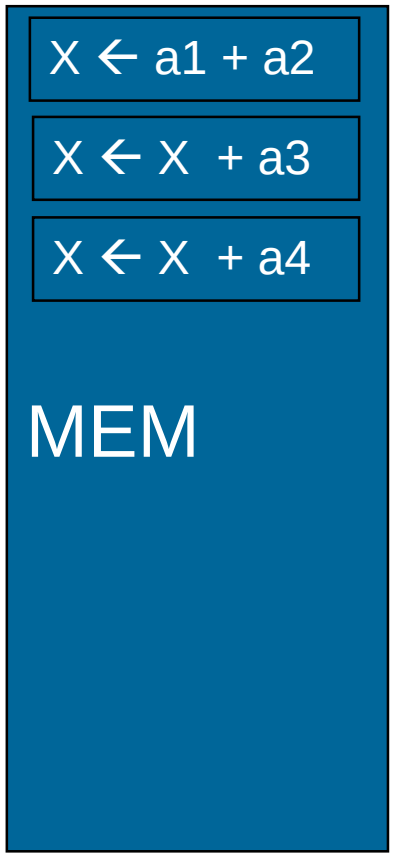
- 计算（加法）
  - ADD src1, src2, dest
- unconditional jump（到第一条指令执行）
  - JMP targetPC
- conditional branch（小于等于N，就跳转）
  - JLE src1, src2, targetPC
- 还有哪些？

重大决策：计算中的operands的位置

- ADD src1, src2, dest
- 原则上，src、dest既可以是CPU中的寄存器（RISC风格），也可以在内存中（CISC风格）
  - 各自优缺点是什么？
- 我们采取RISC风格，这时还需要什么指令？

简单处理器中还需要哪些必要指令？

$$X \leftarrow \sum_{i=1}^{N} a_i$$



- 已有指令
  - ADD src1, src2, dest
  - JMP targetPC
  - JLE src1, src2, targetPC
- 将a_i从memory中读出
  - LOAD addr, dest
- 将X写到memory中
  - STORE src, addr

简单CPU的ISA

- 32位操作
- RISC风格
  - 计算中的src可以是register或者立即数
  - dest只能是register
- 提供4个register供使用
  - R0, R1, R2, R3
- 提供如下指令
  - ADD src1, src2, dest
  - DIV src1, src2, dest
  - JMP targetPC
  - JLE src1, src2, targetPC
  - LOAD addr, dest
  - STORE src, addr

如何实现：

$$X \leftarrow \sum_{i=1}^{N} a_i$$

进一步问题

- 如何对这些指令编码?
- 如何实现/执行这些指令?

进一步问题

- 如何对这些指令编码?
- 如何实现/执行这些指令?

指令编码 1/3

- instruction中都应该包含哪些信息？
  - instruction类型
  - src、dest（立即数，register index）
  - targetPC, addr
  - 并不是所有instruction都包含所有信息
- instruction类型
  - 6个instruction，需要3 bits
    - 000 – ADD, 001 – DIV, 010 – JMP, 011 – JLE, 100 – LOAD, 101 – STORE

ADD src1, src2, dest
DIV src1, src2, dest
JMP targetPC
JLE src1, src2, targetPC
LOAD addr, dest
STORE src, addr

指令编码 2/3

ADD src1, src2, dest
DIV src1, src2, dest
JMP targetPC
JLE src1, src2, targetPC
LOAD addr, dest
STORE src, addr

- src、dest
  - register index, 2 bits
    - 00 – R0, 01 – R1, 10 – R2, 11 – R3
  - 立即数怎么办？是否全部支持32位?
    - ADD 0x00FF1234, 0x11110000, R3
      至少需要多少位?
  - 如何来标记src是register index还是立即数?
- targetPC 、addr
  - JMP targetPC中的targetPC， 以及LOAD/STORE中的addr如果是立即数的话，最多需要30位（为什么？）
  - JLE src1, src2, targetPC中的targetPC是否也可以是30位立即数?
    - JLE 0x00FF1234, 0x11110000, targetPC， 至少需要多少位?
  - 与src相关的问题，JLE src1, src2, targetPC中的src是否可以是立即数?

指令编码 3/3

- 指令长度是可变的，还是不可变的？
  - JMP R0和DIV 0x80000000, 0x12345678, R0的指令长度应该是一样的么？
  - JMP R0中“有用”信息
    - 010　0　00
    - JMP　register　R0
  - DIV 0x80000000, 0x12345678, R0中有用信息
    - 001　1　0x80000000 1　0x12345678 00
    - DIV　imm　0x80000000　imm　0x12345678, R0
- CISC style v.s. RISC style
  - 各自的优势与劣势？

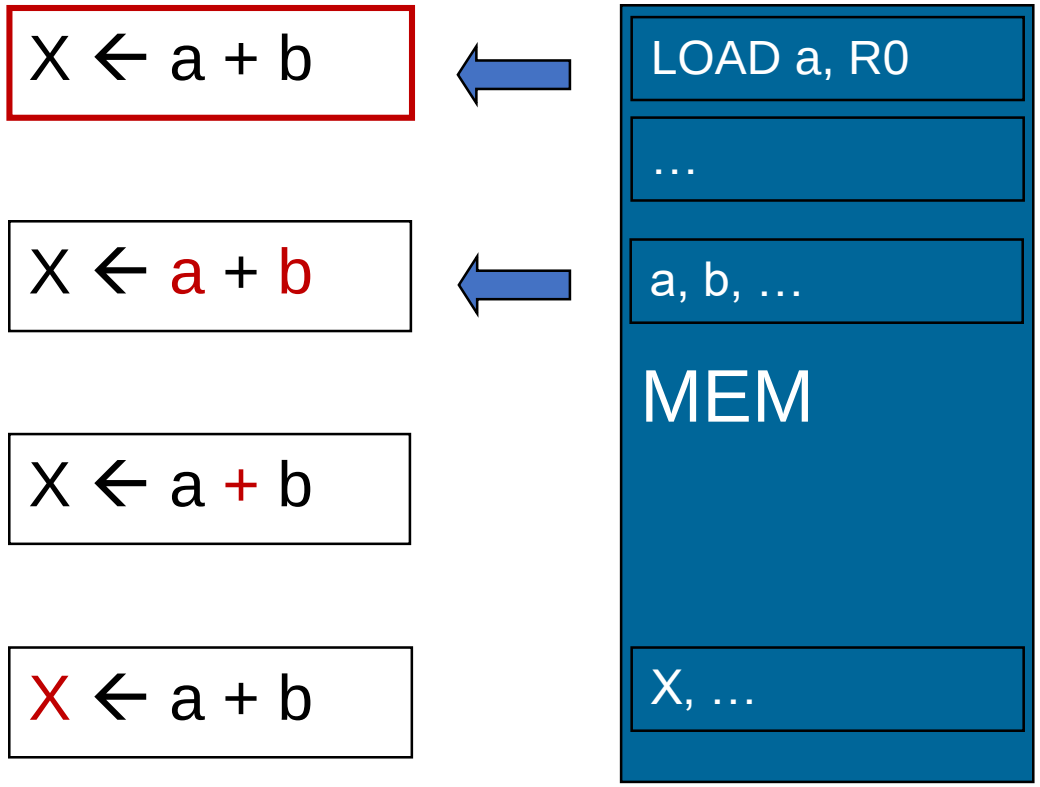
进一步问题

- 如何对这些指令编码?
- 如何实现/执行这些指令?

CPU如何完成一条计算?

X ← a + b

- LOAD a, R0
- LOAD b, R1
- ADD R0, R1, R2
- STORE R2, X

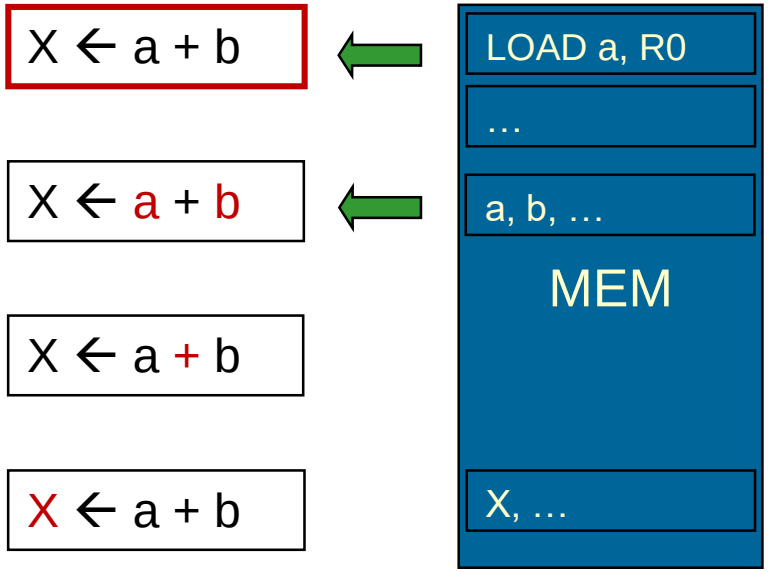


如何实现/执行指令

- 取指 Instruction fetch (IF)
- 译码 Instruction decode (ID)
- 执行 Instruction execute (EX)
- 内存 memory (MEM)
- 回写 write back (WB)

X ← a + b

- LOAD a, R0
- LOAD b, R1
- ADD R0, R1, R2
- STORE R2, X



设计IF级

- 假设已有指令缓冲 instruction buffer

```
CurrInst = InstBuffer[regPC];
```

设计ID级

- 对取得的指令进行译码
- 不同instruction进行不同处理

```
ExecOp = currInst[2:0];

IsToWB = (ExecOP == `ADD) | (ExecOP == `DIV)
         | (ExecOP == `LOAD);

case (ExecOP)
  `ADD: // decoding, 得到Src1, Src2 value, DestRegIndex
  `DIV: // decoding, 得到Src1, Src2 value, DestRegIndex
  `JMP: // decoding, 得到TargetPC value
  `JLE: // decoding, 得到Src1, Src2, TargetPC value
  `LOAD: // decoding, 得到Addr value, DestRegIndex
  `STORE: // decoding, 得到Src1, Addr value
  default: Exception_ID = 1'b1;
endcase
```

ADD src1, src2, dest

DIV src1, src2, dest

JMP targetPC

JLE src1, src2, targetPC

LOAD addr, dest

STORE src, addr

设计EX级

- 进行计算、处理NextPC

```
case (ExecOp)
  `ADD: EX_output = Src1 + Src2;
  `DIV: EX_output = Divider_output;
  `JMP: EX_output = TargetPC;
  `JLE: EX_output = (Src1 <= Src2) ? TargetPC
                                   : (regPC + 1);
  default: // 如何处理EX_output?
endcase

if ((ExecOP == `JMP) | (ExecOP == `JLE))
  NextPC = EX_output;
else
  NextPC = regPC + 1;

Exception_EX = (ExecOp==`DIV) & (src2==0);
```

ADD src1, src2, dest

DIV src1, src2, dest

JMP targetPC

JLE src1, src2, targetPC

LOAD addr, dest

STORE src, addr

设计MEM级


```
if ((ExecOP == `LOAD) | (ExecOP == `STORE)) begin
  if (inIsMemBufferReady) begin
    outLoadEnable = (ExecOP == `LOAD);
    outStoreEnable = (ExecOP == `STORE);
    Exception_MEM = 1'b0;
  end
  else begin
    outLoadEnable = 1'b0;
    outStoreEnable = 1'b0;
    Exception_MEM = 1'b1;
  end
end
else begin
  outLoadEnable = 1'b0;
  outStoreEnable = 1'b0;
  Exception_MEM = 1'b0;
end
```

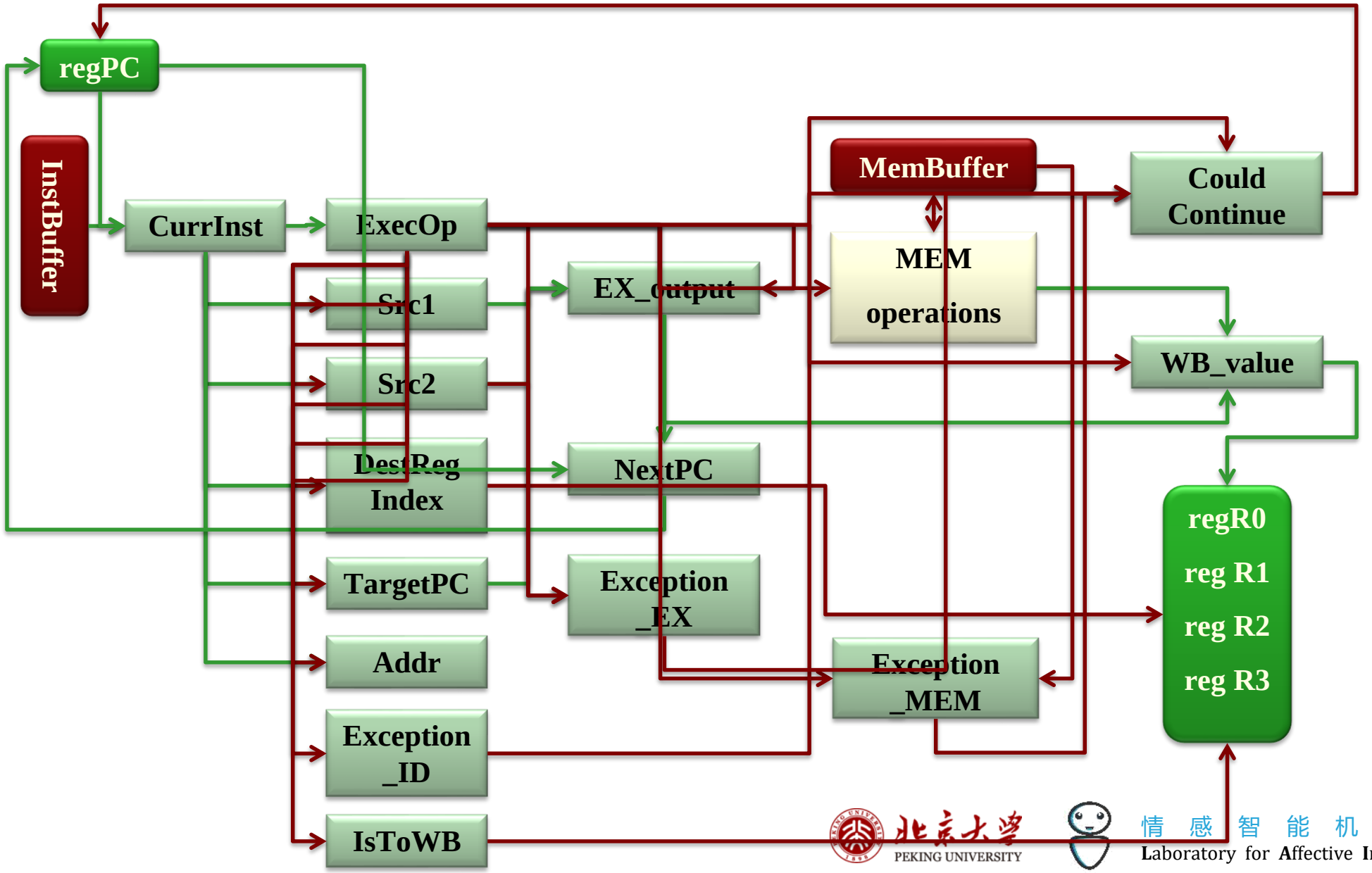
- 假设存在数据缓冲 memory buffer

设计WB级


```
assign CouldContinue = (!Exception_EX) & (!Exception_MEM)
                                       & (!Exception_ID);
assign WB_value = (ExecOp == `LOAD) ? LoadData : EX_output;
always @(posedge CLK)
begin
  if (CouldContinue) begin
    if (IsToWB)
      case (DestRegIndex)
        00: regR0 <= WB_value;
        01: regR1 <= WB_value;
        10: regR2 <= WB_value;
        11: regR3 <= WB_value;
      endcase
    regPC <= NextPC;
  end
end
```

- 如果需要，则更新register
- 判断是否继续、更新PC

简单CPU内部结构


regPC
InstBuffer
CurrInst
ExecOp
Src1
Src2
DestReg Index
TargetPC
Addr
Exception _ID
IsToWB
EX_output
NextPC
Exception _EX
MemBuffer
MEM operations
Exception _MEM
Could Continue
WB_value
regR0
reg R1
reg R2
reg R3

第四课：指令系统

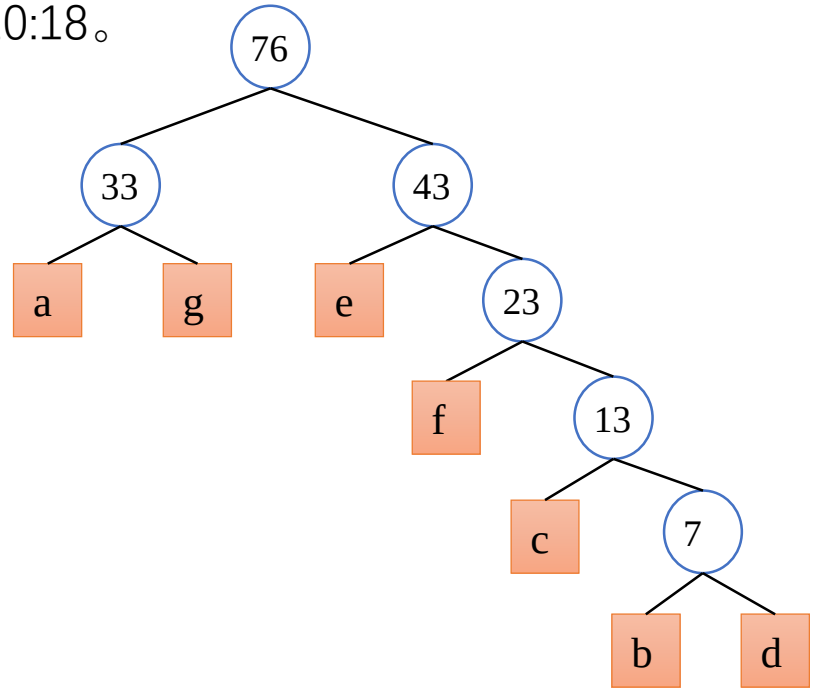
- 数字表示方法
- 设计简单处理器
- 概念与实例

RISC

- 精简指令集计算机
  - MIPS、PowerPC，……
  - 与其相对：CISC复杂指令集计算机 – x86
- RISC的特点
  - 固定的指令长度
  - 采用LOAD、STORE方式访问内存
  - 有限的寻址方式
  - 有限的操作集
- 目标
  - 优化最常用的情景
  - 简化处理器的实现
- 优缺点？

CISC处理器的指令编码

- CISC处理器的指令长度是可变的，指令操作码部分的长度也是可变的
  - 有何好处?
- 如果一个CISC处理器各指令使用的频率不同，应当如何对操作码编码?
  - 设共有7条指令a, b, c, d, e, f, g，使用频率比值为：15:2:6:5:20:10:18。
  - Huffman树
  - a: 00
  - b: 11110
  - c: 1110
  - d: 11111
  - e: 10
  - f: 110
  - g: 01
  - 平均码长是多少？如果采用RISC方案是多少?

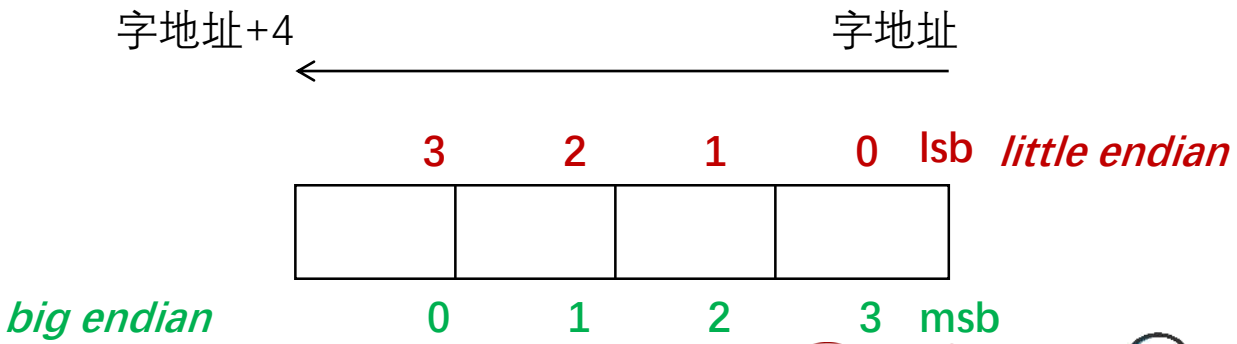


内存：按字节寻址

- 8位的字节很常用，因此大多数系统都在内存中按照字节寻址
  - 注意，于是，对于一个32位系统而言，一个原始含义的字，在内存中的地址均为4的倍数（对齐）
- 对于原始含义的字，如何在内存中存放其各个字节？
  - 大端Big Endian: 最左边的字节（最高位msb）放在字地址上

    IBM 360/370, Motorola 68k, MIPS, Sparc, HP PA
  - 小端Little Endian: 最右面的字节（最低位lsb）放到字地址上

    Intel 80x86, DEC Vax, DEC Alpha (Windows NT)

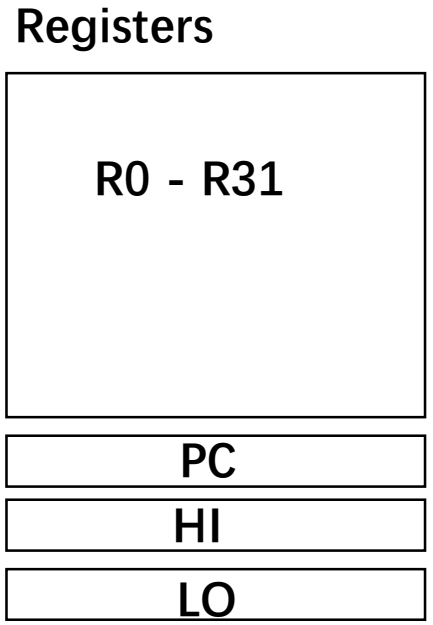


主要寻址方式

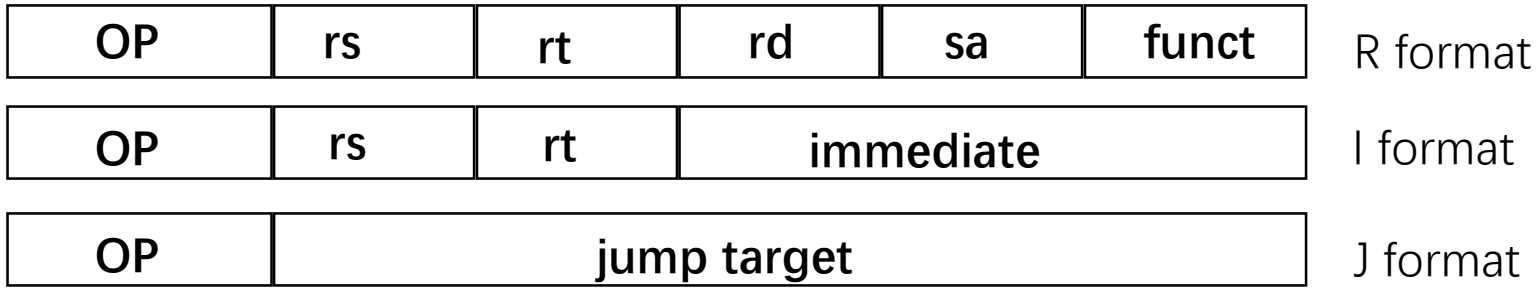
- 立即数寻址方式
  - 所需数据作为立即数编码在指令中
- 寄存器寻址方式
  - 所需数据在某一寄存器中，指令指定这一寄存器
- 直接寻址方式
  - 地址编码在指令之中
- 寄存器间接寻址方式
  - 地址在某一寄存器中，指令中指定这一寄存器
- 基址间接寻址方式（变址寻址方式）
  - 地址为：某一在指令中指定的寄存器中的值+一个在指令中指定的立即数
  - 某些时候，寄存器也可以是PC，或者PC中的前几位
  - 为何需要这样的寻址方式？
    - 转移、数组
- 基址变址寻址方式（两个寄存器）
- 各自优缺点？

MIPS R3000指令集体系结构

- RISC处理器
- 指令类型
  - Load/Store
  - Computational
  - Jump and Branch
  - Floating Point
  - Memory Management
  - Special



3 Instruction Formats: all 32 bits wide

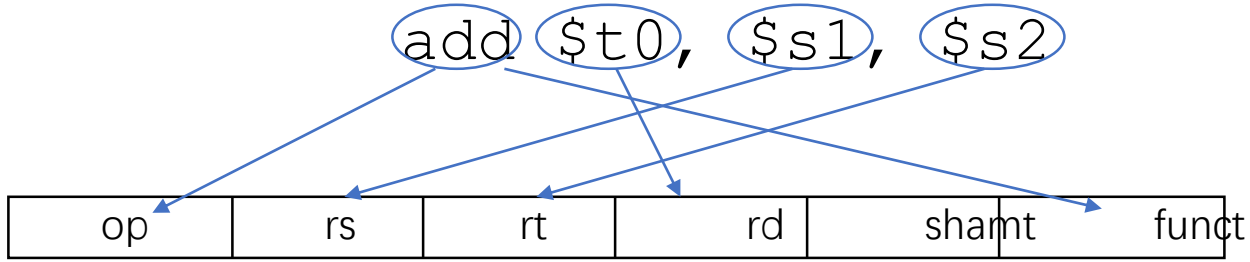


MIPS的寄存器

Name	Register Number	Usage	Preserve on call?
$zero	0	constant 0 (hardware)	n.a.
$at	1	reserved for assembler	n.a.
$v0 - $v1	2-3	returned values	no
$a0 - $a3	4-7	arguments	yes
$t0 - $t7	8-15	temporaries	no
$s0 - $s7	16-23	saved values	yes
$t8 - $t9	24-25	temporaries	no
$gp	28	global pointer	yes
$sp	29	stack pointer	yes
$fp	30	frame pointer	yes
$ra	31	return addr (hardware)	yes

MIPS指令格式 1/3

- 所有指令均为32位长
- R格式指令



op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the second source operand
rd	5-bits	register file address of the result's destination
shamt	5-bits	shift amount (for shift instructions)
funct	6-bits	function code augmenting the opcode

MIPS指令格式 2/3

- I格式指令

```
addi  $sp, $sp, 4      # $sp = $sp + 4
slti $t0, $s2, 15      # $t0 = 1 if $s2<15
```

I format

op	6-bits	opcode that specifies the operation
rs	5-bits	register file address of the first source operand
rt	5-bits	register file address of the result's destination
immediate	16-bit	$+2^{15}-1$ to -2^{15}

MIPS指令格式 3/3

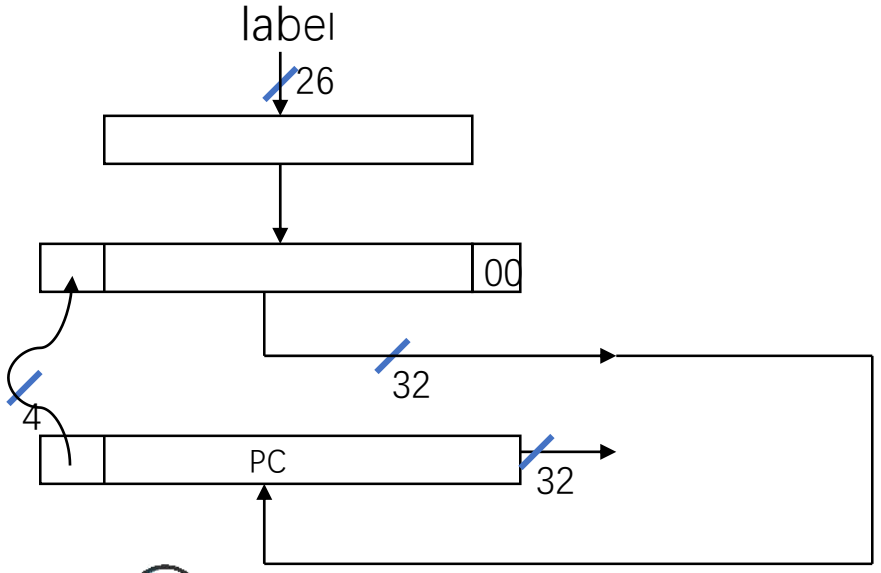
- J指令格式

```
j      lable# go to label
```

J format

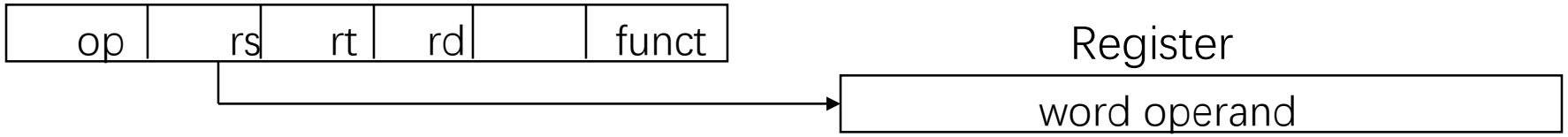
op	6-bits	opcode that specifies the operation
lable	26-bits	$+2^{25}-1$ to -2^{25}

- 思考：如果要跳转到更远处怎么办？
  - 用R指令格式
    - `jr $rs`



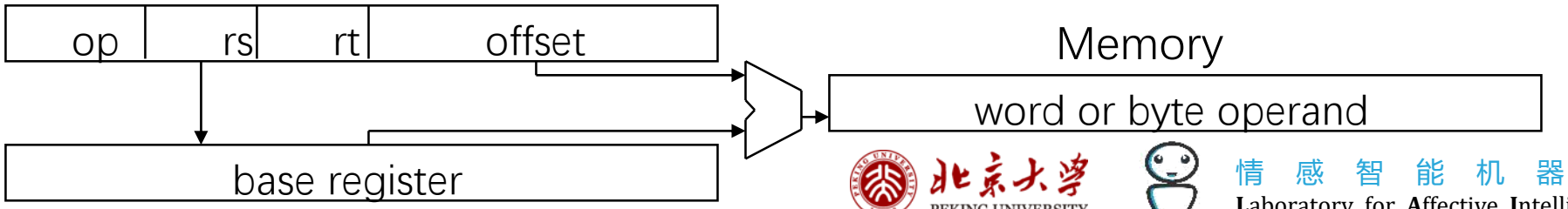
MIPS寻址方式 1/2

- Register addressing（寄存器寻址） – 操作数在寄存器中



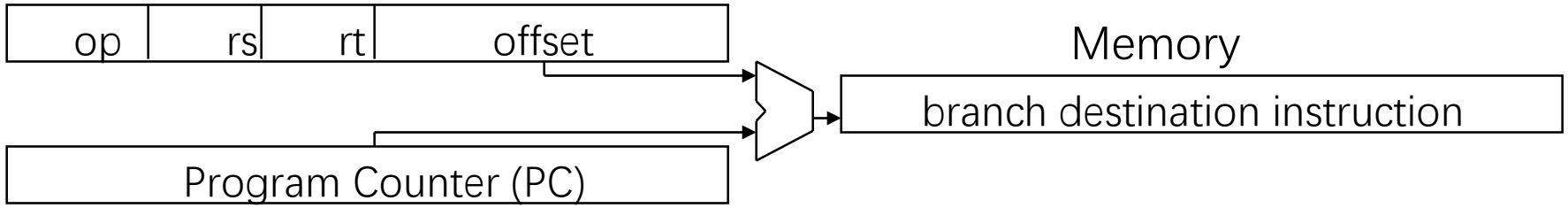
- Immediate addressing（立即数寻址） – 操作数是指令中的一个16位的立即数

- Base (displacement) addressing（基址间接寻址） – 操作数在内存中，内存地址为一个寄存器值与指令中一个16位立即数的和

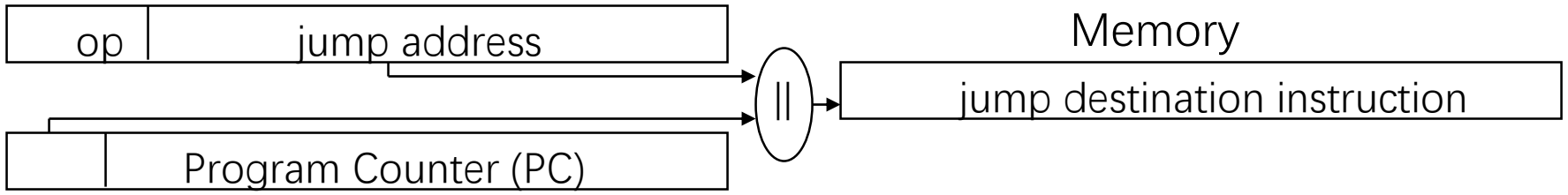


MIPS寻址方式 2/2

- PC-relative addressing （基址间接寻址之一） – （转移）指令的地址是PC与指令中一个16位立即数的和



- Pseudo-direct addressing （基址间接寻址之二） – （跳转）指令的地址是PC的高4位接上指令中的26位立即数



MIPS ISA (part)

Category	Instr	Op Code	Example	Meaning
Arithmetic (R & I format)	add	0 and 32	add $s1, $s2, $s3	$s1 = $s2 + $s3
	subtract	0 and 34	sub $s1, $s2, $s3	$s1 = $s2 - $s3
	add immediate	8	addi $s1, $s2, 6	$s1 = $s2 + 6
	or immediate	13	ori $s1, $s2, 6	$s1 = $s2 v 6
Data Transfer (I format)	load word	35	lw $s1, 24($s2)	$s1 = Memory($s2+24)
	store word	43	sw $s1, 24($s2)	Memory($s2+24) = $s1
	load byte	32	lb $s1, 25($s2)	$s1 = Memory($s2+25)
	store byte	40	sb $s1, 25($s2)	Memory($s2+25) = $s1
	load upper imm	15	lui $s1, 6	$s1 = 6 * 2^{16}
Cond. Branch (I & R format)	br on equal	4	beq $s1, $s2, L	if ($s1==$s2) go to L
	br on not equal	5	bne $s1, $s2, L	if ($s1 !=$s2) go to L
	set on less than	0 and 42	slt $s1, $s2, $s3	if ($s2<$s3) $s1=1 else $s1=0
	set on less than immediate	10	slti $s1, $s2, 6	if ($s2<6) $s1=1 else $s1=0
Uncond. Jump (J & R format)	jump	2	j 2500	go to 10000
	jump register	0 and 8	jr $t1	go to $t1
	jump and link	3	jal 2500	go to 10000; $ra=PC+4

MIPS (RISC) 设计原则

- 简洁性有助于规整化 Simplicity favors regularity
  - 固定长度的指令 — 32位
  - 指令格式数目较少
  - 操作码总是头6位
- 好的设计需要好的妥协 Good design demands good compromises
  - 只有三种指令格式
- 更小的会更快 Smaller is faster
  - 较小的指令集
  - 较少的寄存器
  - 较少的寻址方式
- 使常用的情况变快
  - 算数操作数来自寄存器 (load-store machine)
  - 允许指令包括立即数

总结

- 数字表示方法
- 设计简单处理器
- 概念与实例

- 下一课：存储层次

思考题

1. 一台模型机共有7条指令，各指令的使用频率分别为35%、25%、20%、10%、5%、3%和2%，有8个通用数据寄存器，2个变址寄存器。
   a) 要求操作码的平均长度最短，请设计操作码的编码，并计算所设计操作码的平均长度。
   b) 设计8字长的寄存器-寄存器型指令3条，16位字长的寄存器-存储器型变址寻址方式指令4条，变址范围不小于±127。请设计指令格式，并给出各字段的长度和操作码的编码。
2. 某处理机的指令字长为16位，有双地址指令、单地址指令和零地址指令三类，并假设每个地址字段的长度均为6位。
   a) 如果双地址指令有15条，单地址指令和零地址指令的条数基本相同，问单地址指令和零地址指令各有多少条？并且为这三类指令分配操作码。
   b) 如果要求三类指令的比例大致为1:9:9，问双地址指令、单地址指令和零地址指令各有多少条？并且为这三类指令分配操作码。
3. CISC和RISC的概念及其特点。
4. 简要介绍RISC处理机采用的几项关键技术。