CPU Time=CPU Clock Cycles * Clock Cycle Time= CPU Clock Cycles / Clock Rate
Clock rate = Clock cycles / CPU time
Clock cycles = instruction count x cycle per instruction = (instruction count * CPI) / Clock Rate
Clock Cycles = $\sum_{i=1}^{n} CPI_i * Instruction\ Count_i$
Weighted average CPI = $\sum_{i=1}^{n} CPI_i * (Instruction\ Count_i / instruction\ Count)$
CPU Time = (Instructions/ program) * (Clock Cycles/ instructions) * (seconds/ clock cycle)
T improved = T affected/ improvement factor + T unaffected

A[5] = h + A[3]
lw $t0, 12($s3) # load word
add $t0, $s2, $t0
sw $t0, 20($s3) # store word

```
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: …
```

OP	RS	RT	RD	SHAMT	FUNCT
6	5	5	5	5	6

Jump also in I-format
beq and bne

OP	RS	RT	CONSTANT/ADDRESS
6	5	5	16

0	0000	4	0100	8	1000	c	1100
1	0001	5	0101	9	1001	d	1101
2	0010	6	0110	a	1010	e	1110
3	0011	7	0111	b	1011	f	1111

正的到负的要记得专为相反的 再加一

fact:
addi $sp, $sp, -8 # adjust stack for 2 items
sw $ra, 4($sp) # save return address
sw $a0, 0($sp) # save argument

slti $t0, $a0, 1 # test for n < 1
beq $t0, $zero, L1

addi $v0, $zero, 1 # if so, result is 1
addi $sp, $sp, 8 # pop 2 items from stack
jr $ra. # and return

L1:
addi $a0, $a0, -1 # else decrement n
jal fact # recursive call
lw $a0, 0($sp) # restore original n
lw $ra, 4($sp) # and return address
addi $sp, $sp, 8 # pop 2 items from stack
mul $v0, $a0, $v0 # multiply to get result
jr $ra # and return

C code:
```
int fact (int n)
{
  if (n < 1) return 1;
  else return n * fact(n - 1);
}
```
- Argument n in $a0
- Result in $v0

lb t,off(b) # $t <— Sign-extended byte
lbu t,off(b) # $t <— Zero-extended byte
sb t,off(b) # The byte at off+b <— low-order # byte from register $t.

lh t,off(b) # $t <— Sign-extended halfword
lhu t,off(b) # $t <— zero-extended halfword
sh t,off(b) # Halfword at off+b <— low-order # two bytes from $t.

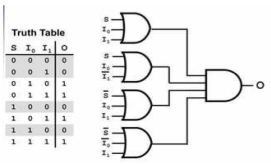
lui Copies 16-bit constant to left 16 bits of rt and Clears right 16 bits of rt to 0

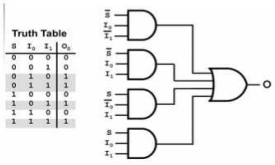
Dynamic linking … Requires procedure code to be relocatable & Automatically picks up new library versions

swap: sll$t1,$a1,2 #$t1=k*4
add $t1, $a0, $t1 # $t1 = v+(k*4) # (address of v[k])
lw $t0, 0($t1) # $t0 (temp) = v[k]
lw $t2, 4($t1) # $t2 = v[k+1]
sw $t2, 0($t1) # v[k] = $t2 (v[k+1])
sw $t0, 4($t1) # v[k+1] = $t0 (temp)
jr $ra # return to calling routine

chapter 3 a)
- NOR: is a 0 if either operand is a 1
- NAND: is a 0 only if both operands are 1
- XOR: is a 1 if the operands are different

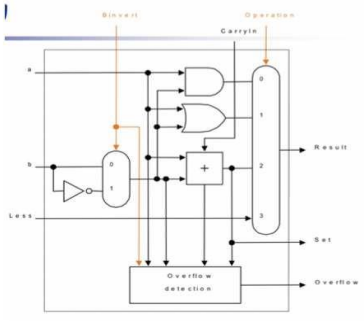
pos





sop

To compute slt A B:
subtract B from A (set binvert and the LS Carry In to 1)
Result for all 1-bit ALUs except the LS should always be 0
Result for the LS 1-bit ALU should be the result bit from the MS 1-bit ALU
LS: Least significant (rightmost) MS: Most significant (leftmost)

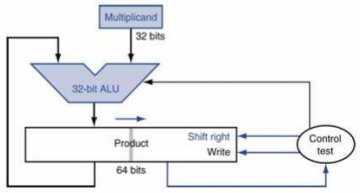


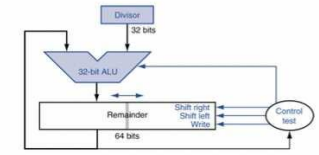

```
avg:    move  $t0, $a0
        move  $t1, $a1
        li    $t2, 0
        la    $t3, const0
        lwc1  $f16, 0($t3)
loop:   beq   $t2, $t1, done
        lwc1  $f17, 0($t0)
        add.s $f16, $f16, $f17
        addi  $t2, $t2, 1
        addi  $t0, $t0, 4
        j     loop
done:   la    $t3, const12
        lwc1  $f17, 0($t3)
        div.s $f0, $f16, $f17

        jr    $ra
```

```
c.lt.s $f12, $f13
bclt   w_anchorage
la     $a0, troy_str
li     $v0, 4
syscall
j      w_done
```

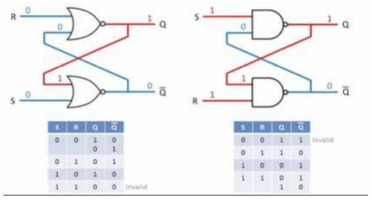
$C_{i+1} = G_i + P_i \cdot C_i$
$G_i = A_i \cdot B_i$
$P_i = A_i + B_i$

Operation	Operand A	Operand B	overflow
X =A+ B	A≥0	B≥0	X <0
X =A+ B	A<0	B<0	X≥0
X=A–B	A≥0	B <0	X <0
X=A–B	A<0	B≥0	X≥0





div $a0, $a1
- mfhi $a2 # reminder to $a2
- mflo $v0 # quotient to $v0



single: 8 bits
double: 11 bits
single: 23 bits
double: 52 bits

S	Exponent	Fraction

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

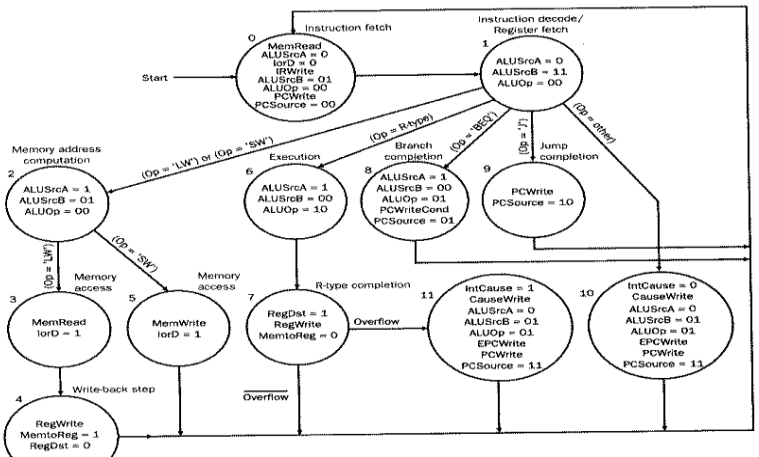
- S: sign bit (0 ⇒ non-negative, 1 ⇒ negative)
- Normalized significand: 1.0 ≤ |significand| < 2.0
  - Always has a leading pre-binary-point 1 bit, so no need to represent it explicitly (hidden bit)
  - Significand is Fraction with the "1." restored
- Exponent: actual exponent + Bias
  - Ensures exponent is unsigned
  - Single: Bias = 127; Double: Bias = 1023

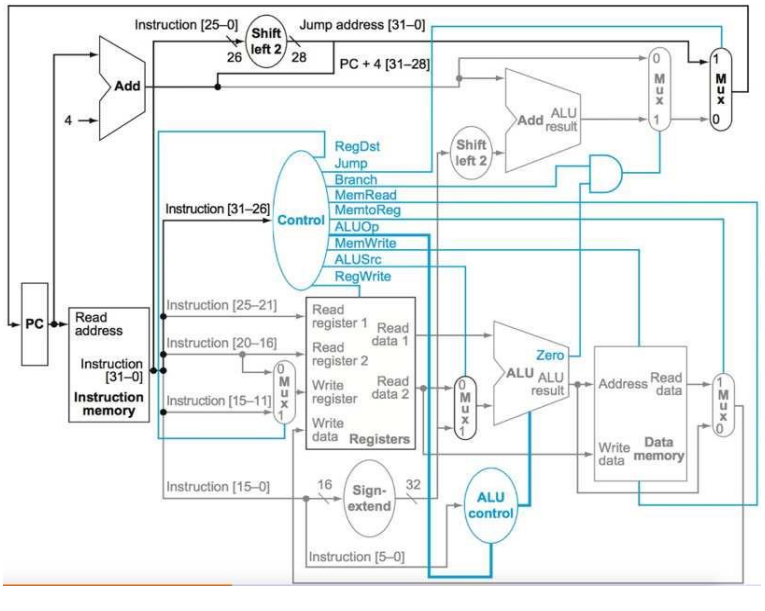
Single precision 32 bit … 6 decimal digits
Double precision 64 bit … 16 decimal digits

Binary … dividing right … multiplying left

Single- and double-precision comparison
e.g. c.xx.s, c.xx.d (xx is eq, lt, le, …) with bclt && bclf

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



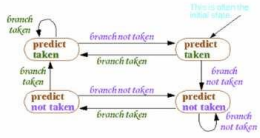


chapter 4 b)
lw $1, addr ...MEM
add $4, $5, $6 ... EX
beq stalled
beq $1, $4, target ...pointing to ID

lw $1, addr... MEM
beq stalled
beq stalled
beq $1, $0, target... pointing to ID

A single prediction bit does not work well with loops
- mispredicts the first & last iterations of a nested loop

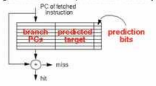
Two-bit branch prediction for loops
- Algorithm: have to be wrong twice before the prediction is changed



2: Branch Target Buffer (BTB)

Cache that stores: the PCs of branches
the predicted target address
branch prediction bits

Accessed by PC address in fetch stage
If hit: address was for *this* branch instruction
fetch the target instruction if prediction bits say taken

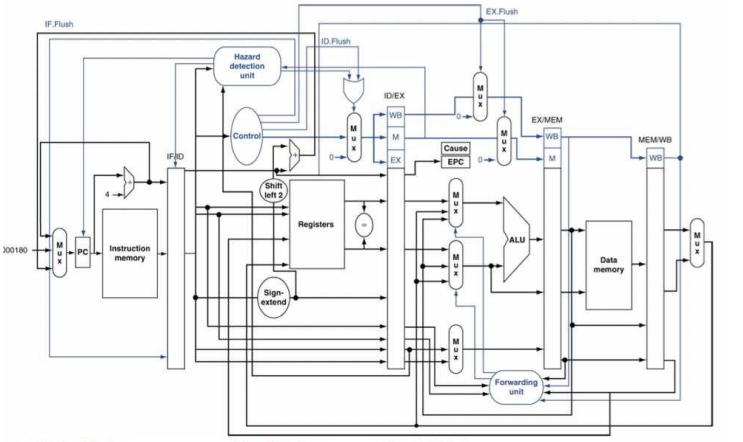


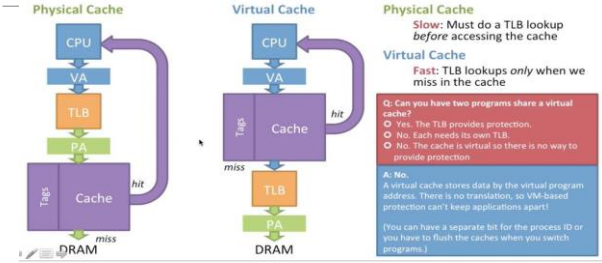
No branch delay if: branch found in BTB
prediction is correct
(assume BTB update is done in the next cycles)

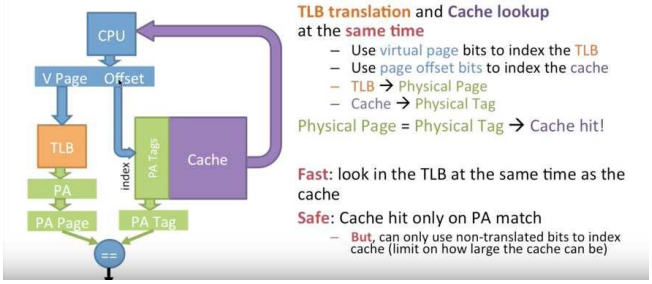
Exception ... CPU ... division by zero and undefined ... system control coprocessor ... save PC o
offending ... mips for Exception Program Counter
Eg. add $1, $2, $1
- Flush add and subsequent instructions
- Set Cause and EPC register values
- Transfer control to handler

Interrupt ... I/O controller ... keyboard

Multiple exception ... overlap ... flush subsequent instruction & precise exception
Imprecise exceptions ... stop & handler
Instruction- level parallelism ... deeper pipeline (shorter clock cycle) & multiple issue (static ..
made by the **compiler** before execution ... issue packets & dynamic... during execution by the
CPU)







TLB translation and **Cache lookup** at the **same time**
- Use virtual page bits to index the TLB
- Use page offset bits to index the cache
- TLB → Physical Page
- Cache → Physical Tag

Physical Page = Physical Tag → Cache hit!

Fast: look in the TLB at the same time as the cache

Safe: Cache hit only on PA match
- But, can only use non-translated bits to index cache (limit on how large the cache can be)

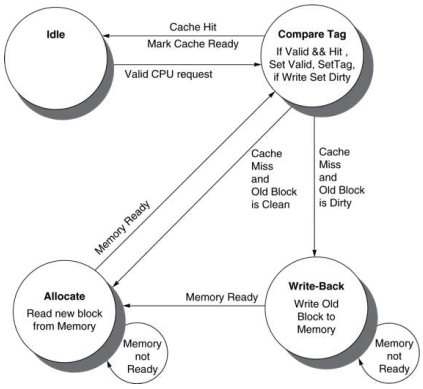
Quiz summary
当新建 array 的时候在 heap 上， function 传入的时候要加&（地址传入）
Echo $? Last process call
0 successfully 1 unsuccessful

```
addi  $sp, $sp, -12
sw    $s0, 8($sp)
sw    $s1, 4($sp)
sw    $s2, 0($sp)

 lw    $ra, 0($sp)
 addi  $sp, $sp, 4
```

For memory operations on full words, an address aligned to the word boundary (i.e., divisible by 0x04) is required. The instruction in question attempts to access memory specifying the effective address which ends in 0x01, i.e., which is not aligned.

Least significant right-most
Most significant left-most

Functionally complete set
Except XOR AND OR



Chapter 5 a) & b)
Recently disk to DRAM
MORE recently DRAM to SRAM

Hit:
Write-Through
Cache immediately writes modified block to memory ... write buffer
Write-back (VM)
先写到 cache, writes back to memory when the block is evicted ... dirty bit
Miss:
Write allocation
Read block into the cache, update the copy in the cache

stall cycles = (num of reads / num of program) * read miss rate * read miss penalty
stall cycles = (num of writes / num of program) * write miss rate * write miss penalty + write buffer stalls

AMAT = Hit time + Miss rate × Miss penalty
Miss penalty = access time / (1/clock rate)

Page fault ... LRU ... reference bit set to 1 -> periodically cleared to 0 -> reference bit to 1 (not used recently)

Compulsory misses (aka cold start misses)
First access to a block
Capacity misses
Due to finite cache size
A replaced block is later accessed again
Conflict misses (aka collision misses)
In a non-fully associative cache
Due to competition for entries in a set
Would not occur in a fully associative cache of the same total size

Cache Coherence Protocols
Migration of data to local caches
- Reduces bandwidth for shared memory

Replication of read-shared data
- Reduces contention for access

Snooping
- Writes broadcast on shared bus

Directory-based
- Each block assigned an ordering point

Use C system calls:
– void *malloc(size_t size) returns a pointer to a chunk of memory which is the size in bytes requested
– void *calloc(size_t nmemb, size_t size) same as malloc but puts zeros in all bytes and asks for the number of elements and size of an element.
– void free(void *ptr) deallocates a chunk of memory. Acts much like the delete operator.
– void *realloc(void *ptr, size_t size) changes the size of the memory chunk pointed to by ptr to size bytes.

$$F = \overline{ABC} + \overline{ABC} + \overline{ABC} + \overline{ABC}$$

\ BC	00	01	11	10
0	0	1	1	0
1	0	1	1	0

~ current directory
Absolute path starting with /
–ls lists file names (like DOS dir command).
–who lists users currently logged in.
–date shows the current time and date.
–pwd print working directory
. for current .. for parent
df: shows what disk holds a directory.
cp [options] source dest //copy
rm [options] names... //remove
– when the file was last changed: ls -l
– when the file was created*: ls -lc
– when the file was last accessed: ls -ul
Files:
–r: allowed to read.
–w: allowed to write.
–x: allowed to execute
• Directories:
–r: allowed to see the names of the files.
–w: allowed to add and remove files.
– x: allowed to enter the directory
Chmod **[ugoa][+-=][rwx]** file
ls * //all files and sub files ... ls -al $HOME
ls –al * // including the directory
ls a* // starting with a
ls *b // ending with b
&> print and sent errors