So far: Resolving Data Dependencies

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a great idea
- Hardware mechanisms
  - Stalling
  - Forwarding
  - Out-of-order execution

1

1

So far: Resolving Control Dependencies

- Software Mechanisms
  - Adding NOPs: requires compiler to insert nops, which also take memory space — not a good idea
  - Delay slot: insert instructions that do not depend on the effect of the preceding instruction. These instructions will execute even if the preceding branch is taken — old RISC approach
- Hardware mechanisms
  - Stalling
  - Branch Prediction
  - Return Address Stack

2

2

Quiz 5 Variant

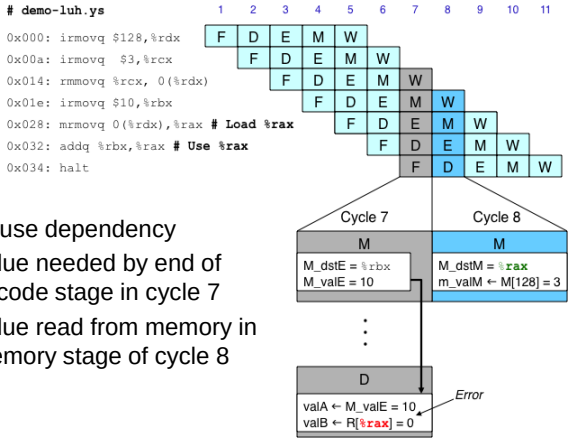
- What does this code do and how long will each loop take in steady state on our 5 stage pipeline? Assume branch is always taken.

```
    irmovl   $5, %edx
    irmovl   $80, %ebx
Loop:
    mrmovl   array_base(%ebx), %eax
    addl     %edx, %eax
    rmmovl   %eax, array_base(%ebx)
    addl     $-4, %ebx
    jne      Loop
```

3

3

Limitation of Forwarding

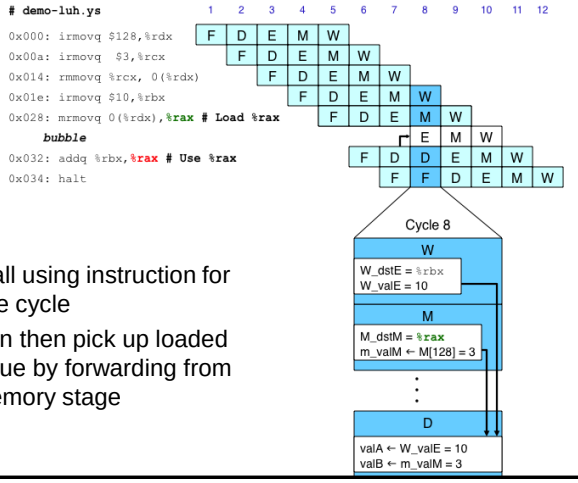


- Load-use dependency
  - Value needed by end of decode stage in cycle 7
  - Value read from memory in memory stage of cycle 8

4

4

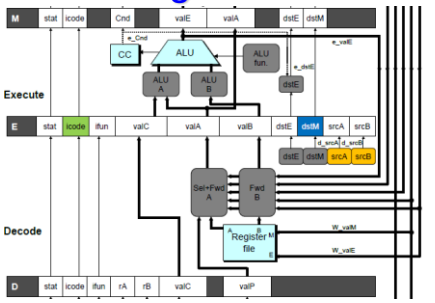
Avoiding Load/Use Hazard



- Stall using instruction for one cycle
- Can then pick up loaded value by forwarding from memory stage

5

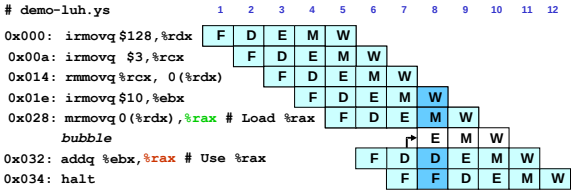
Detecting Load/Use Hazard



Condition	Trigger
Load/Use Hazard	`E_icode in { IMRMOVQ, IPOPQ } && E_dstM in { d_srcA, d_srcB }`

6

Control for Load/Use Hazard



- Stall instructions in fetch and decode stages
- Inject bubble into execute stage

Condition	F	D	E	M	W
Load/Use Hazard	**stall**	**stall**	**bubble**	**normal**	**normal**

7

Branch Prediction

Static Prediction

- Always Taken
- Always Not-taken

Dynamic Prediction

- Dynamically predict taken/not-taken for each specific jump instruction

If prediction is correct: pipeline moves forward without stalling

If mispredicted: kill mis-executed instructions, start from the correct target

8

8

Static Prediction

Observation: Two uses of jumps

- People use jumps to check corner cases. These branches are mostly not taken because corner cases are rare.
- People use jumps to implement loops. These branches are mostly taken because a loop takes multiple iterations.

Strategy:

- Forward jumps (i.e., `if-else`): always predict not-taken
- Backward jumps (i.e., loop): always predict taken

```
  cmpq    %rsi,%rdi
  jle     .corner_case
        <do_A>
  .corner_case:
        <do_B>
        ret
```

Mostly not taken

```
          <before>
.L1:    <body>
        cmpq B, A
        jl .L1
        <after>
```

Mostly taken

Static Prediction

Knowing branch prediction strategy helps us write faster code

- Any difference between the following two code snippets?
- What if you know that hardware uses the always non-taken branch prediction?

```
if (cond) {
    do_A()
  } else {
    do_B()
    }
```

```
if (!cond) {
    do_B()
  } else {
    do_A()
    }
```


Dynamic Prediction

- Simplest idea:
  - If last time taken, predict taken; if last time not-taken, predict not-taken
  - Called 1-bit branch predictor
  - Works nicely for loops

```
for (i=0; i <5; i++) {…}
```

Iteration #1	0	1	2	3	4
Predicted Outcome	N	T	T	T	T
Actual Outcome	T	T	T	T	N

Dynamic Prediction

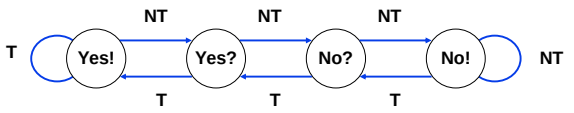
- With 1-bit prediction, we change our mind instantly if mispredict
- Might be too quick. Thus 2-bit branch prediction: we have to mispredict *twice in a row* before changing our mind

```
for (i=0; i <5; i++) {…}
```

Predict with 1-bit	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	
Actual Outcome	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	T	T	T	T	N	
Predict with 2-bit	N	N	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	T	

Example Branch Prediction

- Branch History
  - Encode information about prior history of branch instructions
  - Predict whether or not branch will be taken



- State Machine
  - Each time branch taken, transition to left
  - When not taken, transition to right
  - Predict branch taken when in state Yes! or Yes?

13

More Advanced Dynamic Prediction

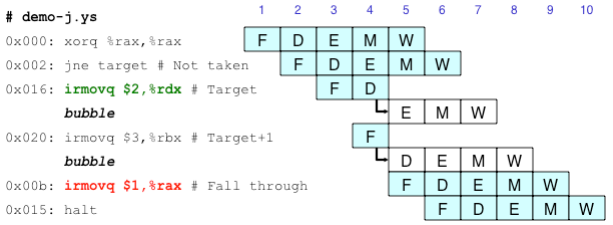
- Look for past histories *across instructions*
- Branches are often correlated
  - Direction of one branch determines another

`cond1` branch not-taken means `(x <=0)` branch taken

```
x = 0
if (cond1) x = 3
if (cond2) y = 19
if (x <= 0) z = 13
```


14

What Happens If We Mispredict?



Cancel instructions when mispredicted

- Detect branch not-taken in execute stage
- On following cycle, replace instructions in execute and decode by bubbles
  - No side effects have occurred yet

15

16

Program Optimization

17

Performance Realities

- *There's more to performance than asymptotic complexity*

- Constant factors matter too!
  - Easily see 10:1 performance range depending on how code is written
  - Must optimize at multiple levels:
    - algorithm, data representations, procedures, and loops
- Must understand system to optimize performance
  - How programs are compiled and executed
  - How modern processors + memory systems operate
  - How to measure program performance and identify bottlenecks
  - How to improve performance without destroying code modularity and generality

19

Generally Useful Optimizations (Machine-Independent)

- Optimizations that you or the compiler should do regardless of processor / compiler

- Code Motion
  - Reduce frequency with which computation performed
    - If it will always produce same result
    - Especially moving code out of loop

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

→

```
    long j;
    int ni = n*i;
    for (j = 0; j < n; j++)
        a[ni+j] = b[j];
```

20

Compiler-Generated Code Motion (-O1)

```
void set_row(double *a, double *b,
   long i, long n)
{
    long j;
    for (j = 0; j < n; j++)
        a[n*i+j] = b[j];
}
```

```
    long j;
    long ni = n*i;
    double *rowp = a+ni;
    for (j = 0; j < n; j++)
        *rowp++ = b[j];
```

```
set_row:
    testq   %rcx, %rcx          # Test n
    jle     .L1                 # If 0, goto done
    imulq   %rcx, %rdx          # ni = n*i
    leaq    (%rdi,%rdx,8), %rdx # rowp = A + ni*8
    movl    $0, %eax            # j = 0
.L3:                            # loop:
    movsd   (%rsi,%rax,8), %xmm0  # t = b[j]
    movsd   %xmm0, (%rdx,%rax,8)  # M[A+ni*8 + j*8] = t
    addq    $1, %rax            # j++
    cmpq    %rcx, %rax          # j:n
    jne     .L3                 # if !=, goto loop
.L1:                            # done:
    rep ; ret
```

21

Share Common Subexpressions (Machine-Independent)

- Reuse portions of expressions
- GCC will do this with –O1

```
/* Sum neighbors of i,j */
up =    val[(i-1)*n + j  ];
down =  val[(i+1)*n + j  ];
left =  val[i*n     + j-1];
right = val[i*n     + j+1];
sum = up + down + left + right;
```

```
long inj = i*n + j;
up =    val[inj - n];
down =  val[inj + n];
left =  val[inj - 1];
right = val[inj + 1];
sum = up + down + left + right;
```

3 multiplications: i*n, (i–1)*n, (i+1)*n

1 multiplication: i*n

```
leaq   1(%rsi), %rax  # i+1
leaq   -1(%rsi), %r8  # i-1
imulq  %rcx, %rsi     # i*n
imulq  %rcx, %rax     # (i+1)*n
imulq  %rcx, %r8      # (i-1)*n
addq   %rdx, %rsi     # i*n+j
addq   %rdx, %rax     # (i+1)*n+j
addq   %rdx, %r8      # (i-1)*n+j
```

```
imulq    %rcx, %rsi  # i*n
addq     %rdx, %rsi  # i*n+j
movq     %rsi, %rax  # i*n+j
subq     %rcx, %rax  # i*n+j-n
leaq     (%rsi,%rcx), %rcx # i*n+j+n
```

22

Reduction in Strength

- Replace costly operation with simpler one
- Shift, add instead of multiply or divide

  `16*x  -->  x << 4`

  - Utility machine dependent
  - Depends on cost of multiply or divide instruction
    - On Intel Nehalem, integer multiply requires 3 CPU cycles
- Recognize sequence of products

```
for (i = 0; i < n; i++) {
  int ni = n*i;
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
}
```

→

```
int ni = 0;
for (i = 0; i < n; i++) {
  for (j = 0; j < n; j++)
    a[ni + j] = b[j];
  ni += n;
}
```

23

Make Use of Registers

- Reading and writing registers much faster than reading/writing memory
- Limitation
  - Compiler not always able to determine whether variable can be held in register
  - Possibility of *Aliasing*
  - See example later

24

Time Scales

- Absolute Time
  - Typically use nanoseconds
    - 10^{-9} seconds
  - Time scale of computer instructions
- Clock Cycles
  - Most computers controlled by high frequency clock signal
  - Typical Range
    - 100 MHz
      - 10^8 cycles per second
      - Clock period = 10ns
    - 2 GHz
      - 2 X 10^9 cycles per second
      - Clock period = 0.5ns

25

Optimizing Compilers

- Provide efficient mapping of program to machine
  - register allocation
  - code selection and ordering (scheduling)
  - dead code elimination
  - eliminating minor inefficiencies
- Don't (usually) improve asymptotic efficiency
  - up to programmer to select best overall algorithm
  - big-O savings are (often) more important than constant factors
    - but constant factors also matter
- Have difficulty overcoming "optimization blockers"
  - potential memory aliasing
  - potential procedure side-effects

26

Limitations of Optimizing Compilers

- Operate under fundamental constraint
  - Must not cause any change in program behavior
    - Except, possibly when program making use of nonstandard language features
  - Often prevents it from making optimizations that would only affect behavior under pathological conditions.
- Behavior that may be obvious to the programmer can be obfuscated by languages and coding styles
  - e.g., Data ranges may be more limited than variable types suggest
- Most analysis is performed only within procedures
  - Whole-program analysis is too expensive in most cases
  - Newer versions of GCC do interprocedural analysis within individual files
    - But, not between code in different files
- Most analysis is based only on *static* information
  - Compiler has difficulty anticipating run-time inputs
- When in doubt, the compiler must be conservative

27