Chapter 4b: Basic Pipelining, cont.

Multicycle Operations

Aggressive Clocking

Vectors

Multicycle Operations

Not all operations complete in one cycle

- Floating-point arithmetic is inherently slower than integer arithmetic
- 2-4 cycles for multiply or add
- 20-50 cycles for divide

Extend DLX pipeline

- EX stage may repeat multiple times
- Multiple function units
  - Not pipelined for now

Handling Multicycle Operations

Four Functional Units

- EX: Integer unit
- E*: FP/integer multiplier
- E+: FP adder
- E/: FP/integer divider

Assume

- EX takes one cycle & all FP units take 4
- Separate integer and FP registers
- All FP arithmetic from FP registers

Worry about

- Structural hazards
- RAW hazards & forwarding
- WAR & WAW between integer & FP ops

FP Instruction Issue

Check for structural hazard

- Wait until function unit is free

Check for RAW data hazard

- Wait until source registers are not used as destinations by instructions in EX

Check for forwarding

- Bypass data from MEM or WB, if necessary

What about overlapping instructions?

- Contention in WB
- Possible WAR and WAW hazards
- Interrupt difficulties

Overlapping Instructions

Contention in WB

Static priority

e.g., FU with longest latency

Instructions stall after issue

WAR hazards

Always read registers at same time

WAW hazards

```
DIVF   F0, F2, F4
SUBF   F0, F8, F10
```

SUBF completes first

(1) Stall SUBF

(2) Abort DIVF's WB

More Multicycle Operations

Problems with Interrupts

```
DIVF   F0, F2, F4
ADDF   F2, F8, F10
SUBF   F6, F4, F10
```

ADDF and SUBF complete before DIVF

Out-of-order completion

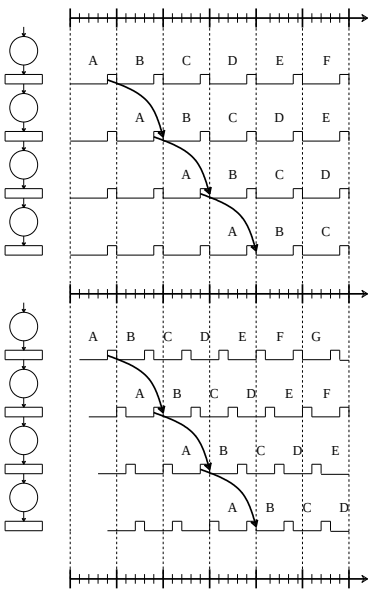
Possible imprecise interrupt

What happens if DIVF generates an exception after ADDF and SUBF complete??

We'll discuss solutions later [Smith88]

Fast Clocking of Pipelines

See [Fawcett, 1976] and [Kunkel & Smith, 1986]

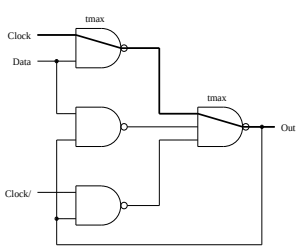


Fast Clocking of Pipelines, cont.

Terms

C_H	clock high
C_L	clock low
AS	intentional clock skew betw. stages
t_{max} (t_{min})	max (min) gate delay
P_{max} (P_{min})	max (min) logic delay

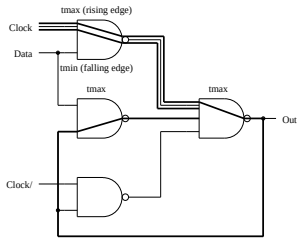
Delay for a Earle latch to propagate data after clock goes high: $2\,t_{max}$



Fast Clocking of Pipelines, cont.

Delay for a Earle latch to hold data:

$$3t_{max} - t_{min}$$



Therefore, in pipeline

$$C_H \geq 3t_{max} - t_{min} + skew\text{–}term\,1$$

Let $skew\text{–}term\,1 \rightarrow 0$

$$C_H \geq 3t_{max} - t_{min} \qquad (1)$$

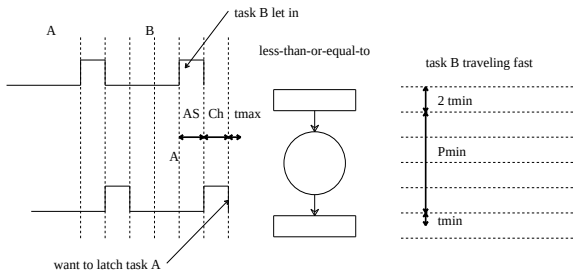
Fast Clocking of Pipelines, cont.

Can't let new data corrupt latch

$$C_H + AS \leq 2t_{min} + P_{min} - skew - 2a - MAX(0,\ t_{max} - t_{min} + skew - 2b)$$

Let $skew\text{–}term's \rightarrow 0$

$$C_H + AS + t_{max} \leq 2t_{min} + P_{min} + t_{min} \qquad (2)$$



Strip Mining

General case: Parameter n

```
      do 10 I = 1, n
         X(i) = a * X(i)
10    continue
```

Strip-mined version (pseudo-code)

```
      low = 1
      VL = (n mod MVL)     /* Odd sized piece */
      do 1 j = 0, (n / MVL)  /* Outer loop */
         do 10 i = low, low+VL-1    /* Length */
            X(i) = a * X(i)
10       continue
         low = low + VL  /* Base of next chunk */
         VL = MVL  /* Reset length to MAX */
1     continue
```

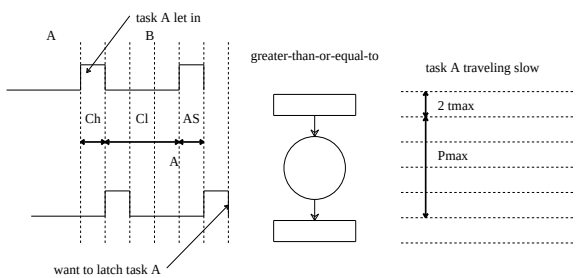

Fast Clocking of Pipelines, cont.

Must wait for correct data

$$C_H + C_L + AS \geq 2t_{max} + P_{max} + skew - 3$$

Let $skew\text{–}term's \rightarrow 0$

$$C_H + C_L + AS \geq 2t_{max} + P_{max} \qquad (3)$$



Fast Clocking of Pipelines, cont.

$$C_H \geq 3t_{max} - t_{min} \quad (1)$$

$$C_H + AS + t_{max} \leq 2t_{min} + P_{min} + t_{min} \quad (2)$$

$$C_H + C_L + AS \geq 2t_{max} + P_{max} \quad (3)$$

Solve (2) for $-AS$

$$-AS \geq C_H - P_{min} - 3t_{min} + t_{max} \quad (2b)$$

Solve (3) for $C_H + C_L$

$$C_H + C_L \geq -AS + 2t_{max} + P_{max} \quad (3b)$$

Put (1) into (2b) and (2b) into (3b):

$$C_H + C_L \geq ((3t_{max} - t_{min}) - P_{min} - 3t_{min} + t_{max}) + 2t_{max} + P_{max}$$

$$C_H + C_L \geq P_{max} - P_{min} + 6t_{max} - 4t_{min} \quad (4)$$

Make look like Kunkel & Smith equation (4)

$$C_H + C_L \geq P_{max} - P_{min} + 5t_{max} - 3t_{min} + (t_{max} - t_{min})$$

Fast Clocking of Pipelines, cont.

Some Implications of equation (4)

$$C_H + C_L \geq P_{max} - P_{min} + 6t_{max} - 4t_{min}$$

Balance gate delays in logic to $n - 2$
(2 levels of logic in Earle latch)

$$C_H + C_L \geq (n-2) \cdot t_{max} - (n-2) \cdot t_{min} + 6t_{max} - 4t_{min}$$

$$C_H + C_L \geq (n+4) \cdot t_{max} - (n+2) \cdot t_{min}$$

Tune logic so that $t_{max} - t_{min} \rightarrow \varepsilon$

$$C_H + C_L \geq (n+4) \cdot t_{max} - (n+2) \cdot (t_{max} - \varepsilon)$$

$$C_H + C_L \geq 2t_{max} + (n+2) \cdot \varepsilon$$

Pipeline throughput determined

latch latency & gate variability!

What are Vector Instructions?

A *vector* is a one-dimensional array of numbers

```
real A(64), B(64), C(64)
```

Many scientific programs operate on vectors of floating point data

```
      do 10 i = 1, 64
         C(i) = A(i) + B(i)
10    continue
```

A *vector instruction* performs an operation on each vector element

```
ADDV C, A, B
```


Why Vector Instructions?

Want Deeper Pipelines, BUT

- Interlock logic complexity grows
- Stalls due to data hazards increase
- Stalls due to control hazards increase
- Hard to issue multiple instrns per cycle (Flynn bottleneck)

Vector instructions allow deeper pipelines

- No *intra*-vector interlock logic
- No *intra*-vector data hazards
- ‘‘Inner’’ loop control hazards eliminated
- Need not issue multiple instrns per cycle
- Vectors have known memory access patterns

Vector Architectures

Vector-Register Machines

- Load/store architecture
- All vector operations use registers (except load/store)
- Multiple ports are cheaper
- Optimized for small vectors

Memory-Memory Vector Machines

- All vectors reside in memory
- Long start-up latency
- Multiple ports are expensive
- Optimized for long vectors

FACT: Most vectors are short

- Early machines were memory-memory (TI ASC, CDC STAR)
- All modern machines use vector-registers

DLXV Architecture

Strongly based on Cray-1 [Russell, 1978]

Extend DLX with vector instructions

- Original DLX is scalar unit
- 32 integer and 32 FP registers

Eight vector registers (V0 - V7)

- 64 double-precision FP EACH (4K bytes total)

Five Vector Functional Units

- FP+, FP*, FP/, integer & logical
- Fully pipelined w/ 2-20 stages

Vector Load/Store Units

- Fully pipelined w/ 10-50 stages

DLXV Architecture, cont.

Vector-Vector Instructions

- Operate on two vectors
- Produce a third vector

```
DO  10 i = 1, 64
    V1(i) = V2(i) + V3(i)
10  CONTINUE


ADDV  V1, V2, V3
```

Vector-Scalar Instructions

- Operate on one vector, one scalar
- Produce a third vector

```
DO  10 i = 1, 64
    V1(i) = F0 + V3(i)
10  CONTINUE


ADDSV  V1, F0, V3
```


DLXV Architecture, cont.

Vector Load/Store Instructions

- Load/Store a vector from memory into a vector register
- Operates on contiguous addresses

```
LV  V1, R1   ; V1[i] = M[R1 + i]
SV  R1, V1   ; M[R1 + i] = V1[i]
```

Load/Store Vector with Stride

- Vectors not always contiguous in memory
- Add *non-unit stride* on each access

```
LVWS V1, (R1, R2)  ; V1[i] = M[R1 + i*R2]
SVWS (R1, R2), V1  ; M[R1 + i*R2] = V1[i]
```

Vector Load/Store Indexed

- Indirect accesses through an index vector

```
LVI V1, (R1+V2)  ; V1[i] = M[R1 + V2[i]]
SVI (R1+V2), V1  ; M[R1 + V2[i]] = V1[i]
```

DLXV Architecture, cont.

Double-precision A*X Plus Y (DAXPY):

```
      do 10 i = 1, 64
         Y(i) = a * X(i) + Y(i)
10    continue

      LD     F0, a
      LV     V1, Rx
      MULTSV V2, F0, V1
      LV     V3, Ry
      ADDV   V4, V2, V3
      SV     Ry, V4
```

6 DLXV instrns instead of 600 DLX instrns

- Remember: MIPS means "Meaningless Indicator of Performance"
- MFLOPS is better measure

Not All Vectors are 64 Elements Long

Vector length register (VLR)

- Controls length of vector operations
- $0 < VLR \leq MVL = 64$

```
      do 10 i = 1, 100
         X(i) = a * X(i)
10    continue

      LD     F0, a
      MOVI2S VLR, 36     /* 100 - 64 */
      LV     V1, Rx
      MULTSV V2, F0, V1
      SV     Rx, V2
      ADD    Rx, Rx, 36
      MOVI2S VLR, 64
      LV     V1, Rx
      MULTSV V2, F0, V1
      SV     Rx, V2
```

Use *strip mining* for:

```
      do 10 i = 1, n
             X(i) = a * X(i)
      10  continue
```


More Options

Use *vector mask register* for vectorizing

```
      do 10 i = 1, 64
         if A(i) < 0.0 then A(i) = -A(i)
10    continue
```

Use *chaining* (vector register bypass) for RAWs

```
      MULTV  V1, --, --
      ADDV   --, V1, --
```

Use *gather/scatter* for sparse matrices

```
      do 10 i = 1, 64
         A(K(i) = A(K(i)) + C(M(i))
10    continue
```

FINAL WARNING: Make scalar unit fast!

- Amdahl's law
- CRAY-1 was the fastest *scalar* computer