CPU-aware Basic Blocks

Some Methods

- Loop Unrolling
- Code Scheduling
- SW Pipelining
- Scalar Replacement
- Skewing
- Vectorization

Readings

- hugue_unrolling
- H&P 3e Ch4.1-4.3
- P&H 4e Ch4.10
- C&F&P Section 5.2

Background

- Dependencies
- Renaming
- Branch Prediction

Hardware Models

- Assembly language
- Standard 5-stage pipeline
- Dual issue 5-stage pipeline
- Long FP times
- Superscalar with FP

Basic Block Basics

Basic Block ⇔ is code that has one entry point (i.e., no code within it is the destination of a jump instruction), one exit point and no jump instructions contained within it. Basic blocks are usually the basic unit to which compiler optimizations are applied.

Loop Optimizations ⇔ Most execution time of high performance program is spent on nested loops of basic blocks.

- Optimizations on loops and basic blocks generally provide most of the benefit that can be extracted at the code level (as opposed to algorithm)
- These ideas will be extended to parallel processing
- We've already done some of this: loop interchange, blocking

Previously → Transformations for memory optimization (B&O Ch. 6)

→ Enable compiler to make optimizations (B&O Ch. 5)

Now → See what's happening w.r.t. real datapaths

→ Although a bit simpler than i7: based on 5 stage pipe w/ 2x superscalar & multiple FP units

→ Additional program optimization methods

→ Software pipelining, skewing, etc.

Loop Unrolling

```
// simple loop example
for(int i=0;i<1000;i++)
  a[i] = b[i] + c[i];
```

ASSEMBLY LANGUAGE VERSION

assume R1 contains the base address for the 'a' array
R2 has the base address for the 'b' array and
R3 has the base address for the 'c' array
R4 starts at 1000

```
Loop:   LW      R5, 0(R2)       ; element of b
        LW      R6, 0(R3)       ; element of c
        ADD     R7, R6, R5      ; make next a
        SW      0(R1), R7       ;
        ADDI    R1, R1, #4      ;
        ADDI    R2, R2, #4      ; increment addresses
        ADDI    R3, R3, #4      ;
        SUBI    R4, R4, #1      ; decrement loop var
        BNEZ    R4, Loop        ;
```

Note: most instructions are loop manipulation!

hugue

```
// UNROLL once
for(int i=0;i<1000;i=i+2) {
  a[i] = b[i] + c[i];
  a[i+1] = b[i+1] + c[i+1];
}

;; Assembly Language version
Loop:    LW      R5, 0(R2)        ; element of b
         LW      R6, 0(R3)        ; element of c
         ADD     R7, R6, R5       ; make next a
         SW      0(R1), R7        ;
         LW      R5, 4(R2)        ; next element of b
         LW      R6, 4(R3)        ; next element of c
         ADD     R7, R6, R5       ; make next a
         SW      4(R1), R7        ;
         ADDI    R1, R1, #8       ;
         ADDI    R2, R2, #8       ; increment addresses
         ADDI    R3, R3, #8       ;
         SUBI    R4, R4, #2       ; decrement loop var
         BNEZ    R4, Loop         ;
```

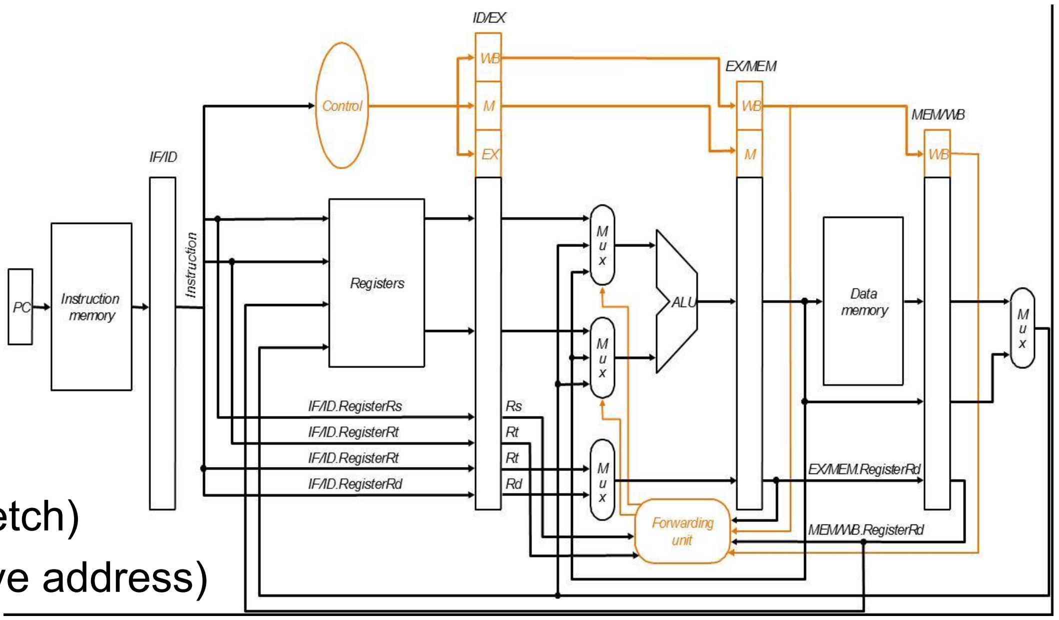
Note: use of displacement mode gets offset for free -- it's part of the instruction.

Note: the more unrolling, the less important the loop overhead. But there is a trade off of space for time, which can get important if the code might not fit in L1 cache.

Note: a bad compiler will make this code worse. A good compiler might unroll for you, especially if the loop index is a constant (like here) and the basic block is trivial (like here).

hugue

Code Interaction with the CPU



→ 5 stages:

IF (instruction fetch)

ID (instruction decode, register operand fetch)

EX (execute operation or compute effective address)

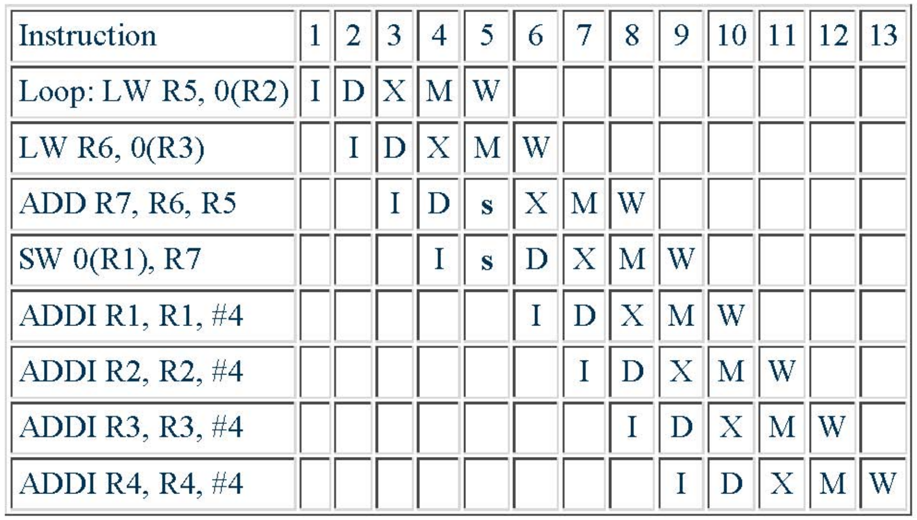
MEM (memory reference if there is one)

WB (write back to register file)

→ forwarding

```
Consecutive arithmetic instructions are OK (integer add/sub & logicals)
EX:  ADD  R1,R2,R3    ; this works --
     ADD  R4,R1,R1    ;  -- R1 gets forwarded in time

arithmetic after a LW requires a stall
EX:   LW     R1,4(R4)     ; this requires a stall to get --
      ADD    R4,R1,R1     ;   -- the data here for correct execution

SW after a LW is OK.  (So is SW after arithmetic)
EX:   LW    R1,4(R4)      ;
      SW    R1,8(R8)      ;  this is OK
```

Map code to 5-stage pipeline. How long does it take? Baseline CPI = 1 (e.g., one instruction issue per cycle). Then count losses due to stalls.



Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13
Loop: LW R5, 0(R2)	I	D	X	M	W								
LW R6, 0(R3)		I	D	X	M	W							
ADD R7, R6, R5			I	D	s	X	M	W					
SW 0(R1), R7				I	s	D	X	M	W				
ADDI R1, R1, #4						I	D	X	M	W			
ADDI R2, R2, #4							I	D	X	M	W		
ADDI R3, R3, #4								I	D	X	M	W	
ADDI R4, R4, #4									I	D	X	M	W

One iteration takes 9 issue cycles. **CPI = 9/8**
The third instruction "ADD R7, R6, R5" has to stall for R6.

However with simple rescheduling we can remove the stall:

```
; Original code
Loop: LW R5, 0(R2)
   LW   R6, 0(R3)
   ADD  R7, R6, R5
; stall happens here
   SW   0(R1), R7
   ADDI R1, R1, #4
   ADDI R2, R2, #4
   ADDI R3, R3, #4
   SUBI R4, R4, #1
   BNEZ R4, Loop
```

```
; Rescheduled code
Loop:   LW     R5, 0(R2)      ; element of b
        LW     R6, 0(R3)      ; element of c
        ADDI   R2, R2, #4     ; increment address
        ADD    R7, R6, R5     ; make next a
        SW     0(R1), R7      ; CAN'T MOVE THIS ONE
        ADDI   R1, R1, #4     ; CAN'T MOVE THIS ONE
        ADDI   R3, R3, #4     ;
        SUBI   R4, R4, #1     ; decrement loop var
        BNEZ   R4, Loop       ;
```

hugue

One iteration now takes 8 issue cycles. Now, **CPI = 1**

```
;; Original Version
Loop: LW R5, 0(R2)
    LW   R6, 0(R3)
    ADD  R7, R6, R5
    SW   0(R1), R7
    LW   R5, 4(R2)
    LW   R6, 4(R3)
    ADD  R7, R6, R5
    SW   4(R1), R7
    ADDI R1, R1, #8
    ADDI R2, R2, #8
    ADDI R3, R3, #8
    SUBI R4, R4, #2
    BNEZ R4, Loop
```

```
;; Scheduled Version
Loop:   LW      R5, 0(R2)     ; element of b
        LW      R6, 0(R3)     ; element of c
        LW      R8, 4(R2)     ; next element of b
        ADD     R7, R6, R5    ; make next a
        LW      R9, 4(R3)     ; next element of c
        SW      0(R1), R7
        ADD     R10, R8, R9   ; make next a + 1
        SW      4(R1), R10
        ADDI    R1, R1, #8
        ADDI    R2, R2, #8    ; increment addresses
        ADDI    R3, R3, #8
        SUBI    R4, R4, #2    ; decrement loop var
        BNEZ    R4, Loop
```

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Loop: LW R5, 0(R2)	I	D	X	M	W											
LW R6, 0(R3)		I	D	X	M	W										
LW R8, 4(R2)			I	D	X	M	W									
ADD R7, R6, R5				I	D	X	M	W								
LW R9, 4(R3)					I	D	X	M	W							
SW 0(R1), R7						I	D	X	M	W						
ADD R10, R8, R9							I	D	X	M	W					
SW 4(R1), R10								I	D	X	M	W				
ADDI R1, R1, #8									I	D	X	M	W			
ADDI R2, R2, #8										I	D	X	M	W		
ADDI R3, R3, #8											I	D	X	M	W	
SUBI R4, R4, #2												I	D	X	M	W

Map unrolled code to 5-stage pipeline.
Note that the more unrolling, the more flexibility, the easier it is to schedule instructions to prevent stalls.

CPI = 1 (still)
Much more interesting is that each iteration now takes 6 cycles rather than 8 (or 9).

What about the control hazard?

```
Loop:   LW      R5, 0(R2)           ; element of b
        LW      R6, 0(R3)           ; element of c
        LW      R8, 4(R2)           ; next element of b
        ADD     R7, R6, R5          ; make next a
        LW      R9, 4(R3)           ; next element of c
        ADD     R10, R8, R9         ; make next a + 1
        SW      0(R1), R7           ;
        SW      4(R1), R10          ;
        ADDI    R1, R1, #8          ;
        ADDI    R2, R2, #8          ; increment addresses
        ADDI    R3, R3, #8
        SUBI    R4, R4, #2          ; decrement loop var
        BNEZ    R4, Loop
        delay slot stall        ; NOP gets executed here
                                ; before transfer to Loop:
```

With no branch prediction, we need to stall at least one cycle every time through except the last. This would be an additional (very good) argument to continue unrolling (to amortize that stall further). But modern processors *do* have good predictors, esp. for simple loops, so should not be an issue. But for now, *two issues*:

1. **Data hazard for BNEZ input operand.** Depending on the implementation of the 5-stage pipeline, the comparison may be done in ID or EX. *The models in this talk use both, so need to check.* But if in ID, then there must be a stall if the conditional branch follows an arithmetic instruction that generates the operand.

2. **Delay slot.** Here we assume:

<u>**delay slot**</u> ⇔ the instruction following a branch or jump is executed unconditionally while the branch is being evaluated. It is ALWAYS executed after the branch instruction. If this is inconvenient, then the delay slot is filled with a NOP.

What about running out of registers?

```
;; Scheduled Version
Loop:   LW      R5, 0(R2)
        LW      R6, 0(R3)
        LW      R8, 4(R2)
        ADD     R7, R6, R5
        LW      R9, 4(R3)
        SW      0(R1), R7
        ADD     R10, R8, R9
        SW      4(R1), R10
        ADDI    R1, R1, #8
        ADDI    R2, R2, #8
        ADDI    R3, R3, #8
        SUBI    R4, R4, #2
        BNEZ    R4, Loop
```

```
;; Modify to reuse registers
Loop:   LW      R5, 0(R2)
        LW      R6, 0(R3)
        ADDI    R2, R2, #8
        ADD     R7, R6, R5
        SW      0(R1), R7
        LW      R5, -4(R2)   ; change offset
        LW      R6, 4(R3)
        ADDI    R3, R3, #8
        ADD     R7, R6, R5
        SW      4(R1), R7
        ADDI    R1, R1, #8
        SUBI    R4, R4, #2
        BNEZ    R4, Loop
```

In Hugue's code, the unrolled version uses additional registers: R8, R9, R10.
We might assume that each unroll would require and additional three registers.
Since registers are finite, running out of registers would appear to be a limiting factor in the amount of unrolling that you can do.
But it doesn't need to be (see modified version). ***Often possible to reuse registers.***
Note: whether you run out of registers has less to do with number of times you unroll and more to do with instruction latency and dependences.

hugue

More on this later -- you may need to schedule instructions across iterations.

Data Dependence

<u>Data Dependence</u> ⇔ An ordering constraint. That is, an order of operand usage such that changing that ordering will cause incorrect program execution.

- Data dependences, and thus data hazards, come in three flavors (not all of which apply to the five stage pipeline):
  - Read after Write (RAW)
  - Write after Write (WAW)
  - Write after Read (WAR)

  Read after Read (RAR) doesn't cause problems since Read does not change state

Note: within a basic block data dependences are obvious. D.D. becomes a complex and challenging issue when there are dependences across iterations.

RAW Dependence

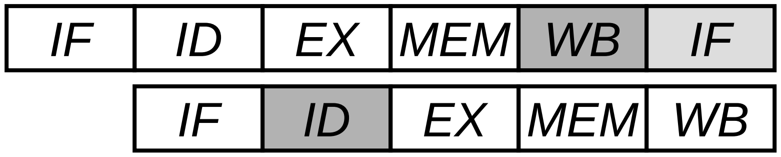
<u>RAW Dependence</u> ⇔ A later instruction reads an operand that was written by a previous instruction

<u>RAW Hazard</u> ⇔ The read occurs before the write

The dependence:

```
ADD  R1,R2,R3       #  the dependence
SUB  R5,R1,R4
```

The pipeline hazard:



WAW Dependence

WAW Dependence ⇔ A later instruction writes to an operand that was written by a previous instruction

WAW Hazard ⇔ The second write occurs before the first write

The dependence:

LW **R1**,100(R4) # the dependence

ADD **R1**,R2,R3

The pipeline hazard (assume memory access takes three cycles):

IF	*ID*	*EX*	*MEM*	*MEM*	*MEM*	*WB*
	IF	*ID*	*EX*	*MEM*	*WB*	

Note: WAW hazards occur in reasonable pipelines, but not in the simple five stage pipeline.

WAR Dependence

<u>WAR Dependence</u> ⇔ A later instruction reads from an operand that was written by a previous instruction

<u>WAR Hazard</u> ⇔ The read occurs before the first write

The **dependence**:

```
DIV    R1,R2,R3          #  the dependence
ADD    R2,R4,R5
```

The pipeline **hazard** (assume DIV must wait for div unit to be available):

IF	*wait*	*wait*	*wait*	*wait*	*wait*	*wait*	*ID*	*EX*	*MEM*	*WB*
	IF	*ID*	*EX*	*MEM*	*WB*	*IF*				

Note: WAR hazards are only likely to occur in datapaths where there is instruction reordering

For now …

Deal with hazards →

With respect to instruction **reordering** *(i.e., change order of instruction issue)*:

RAW → **never** do it. These are REAL hazards.

WAW/WAR → **Can** do it with renaming.

Assume that the code is correct (wrt instruction serialization assumption)

With respect to **pipeline**:

RAW → assume that the HW will take care of these with forwarding and stalls.

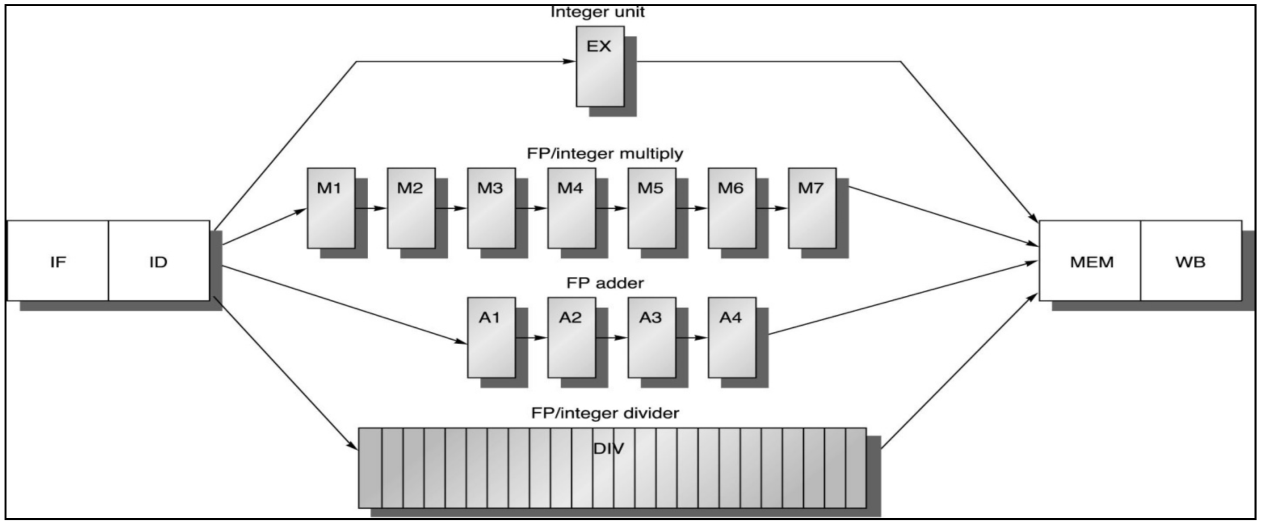
WAW/WAR → assume that the HW will take care of these with internal renaming. If we run out of registers, this is a problem. (But I thought this wasn't a problem? It can be with high latency operations.)

Let’s Look at FP instructions

Model: to start, similar to 5 stage pipeline, but with variable latency in the EX stage (> 1 cycle).

Note: “Latency” means extra delay, or gap, cycles.

Instruction Producing Result	Instruction Using Result	Latency in clock cycles
FP ALU op	Another FP ALU op	3
FP ALU op	Store double	2
Load double	FP ALU op	1
Load double	Store double	0
Int ALU op	Branch	1



H&P 3e

```
for (i = 1000; i > 0; i=i-1)  // x is a double array
  x[i] = x[i] + s;


Loop:   L.D     F0,0(R1)        ; F0=array element
        ADD.D   F4,F0,F2        ; add scalar in F2
        S.D     F4,0(R1)        ; store result
        DADDUI  R1,R1,#-8       ; decrement pointer
        BNE     R1,R2,Loop      ; branch R1!=R2
```

Use FP model to find number of stalls (assumes branch test in ID)

```
                                  Clock Cycle Issued
Loop:   L.D     F0,0(R1)                1
        stall                           2
        ADD.D   F4,F0,F2                3
        stall                           4
        stall                           5
        S.D     F4,0(R1)                6
        DADDUI  R1,R1,#-8               7
        stall                           8
        BNE     R1,R2,Loop              9
        stall                           10
```

H&P 3e

```
original code                         Clock Cycle Issued
Loop:   L.D     F0,0(R1)                     1
        stall                                2
        ADD.D   F4,F0,F2                     3
        stall                                4
        stall                                5
        S.D     F4,0(R1)                     6
        DADDUI  R1,R1,#-8                    7
        stall                                8
        BNE     R1,R2,Loop                   9
        stall                                10
```

Schedule code to remove stalls

```
                                      Clock Cycle Issued
Loop:   L.D     F0,0(R1)              1
        DADDUI  R1,R1,#-8             2  move here
        ADD.D   F4,F0,F2              3
        stall                         4
        BNE     R1,R2,Loop            5  Delayed branch
        S.D     F4,8(R1)              6  alter disp (8) and move here
```

Now 6 cycles rather than 10
But still, half of the work is overhead.
Answer: unroll

H&P 3e

```
// original code
for (i = 1000; i > 0; i=i-1)
  x[i] = x[i] + s;


Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDUI  R1,R1,#-8
        BNE     R1,R2,Loop
```

```
;; Unroll loop four times (unscheduled)

Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)
        L.D     F6,-8(R1)
        ADD.D   F8,F6,F2
        S.D     F8,-8(R1)
        L.D     F10,-16(R1)
        ADD.D   F12,F10,F2
        S.D     F12,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F16,F14,F2
        S.D     F16,-24(R1)
        DADDUI  R1,R1,#-32
        BNE     R1,R2,Loop
        delay slot
```

```
Time: 28 clock cycles (Ugh!)
14 issues
  4 stalls for L.D/ADD.D (4x1)
  8 stalls for ADD.D/S.D (4x2)
  1 stall for DADDUI/BNE
  1 stall for delay slot
```

H&P 3e

```
; unscheduled unrolled code
Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)
        L.D     F6,-8(R1)
        ADD.D   F8,F6,F2
        S.D     F8,-8(R1)
        L.D     F10,-16(R1)
        ADD.D   F12,F10,F2
        S.D     F12,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F16,F14,F2
        S.D     F16,-24(R1)
        DADDUI  R1,R1,#-32
        BNE     R1,R2,Loop

Time: 28 clock cycles (Ugh!)
       CPI = 2
14 issues
  4 stalls for L.D/ADD.D (4x1)
  8 stalls for ADD.D/S.D (4x2)
  1 stall for DADDUI/BNE
  1 stall for delay slot
```

```
; Schedule the unrolled code

Loop:   L.D     F0,0(R1)
        L.D     F6,-8(R1)
        L.D     F10,-16(R1)
        L.D     F14,-24(R1)
        ADD.D   F4,F0,F2
        ADD.D   F8,F6,F2
        ADD.D   F12,F10,F2
        ADD.D   F16,F14,F2
        S.D     F4,0(R1)
        S.D     F8,-8(R1)
        DADDUI  R1,R1,#-32
        S.D     F12,16(R1)
        BNE     R1,R2,Loop
        S.D     F16,8(R1)

Time: 14 clock cycles (!) ☺
       CPI = 1
Note: unrolling exposes more
  code to scheduling
Note: code movement may require
  adjusting instructions
```

H&P 3e

What did we do?

1. Determine that it is legal to move S.D after DADDUI and BNE. Adjust S.D offset.
2. Determine that iterations are independent, which improves chance that unrolling is beneficial
3. Use different registers for each (former) iteration.
4. Eliminate extra tests and branches
5. Determine that LDs and STs from different iterations can be interchanged. This means that you have to figure out the relative memory addresses.
6. Schedule the code preserving the remaining dependencies.

See H&P 3e for details.

H&P 3e

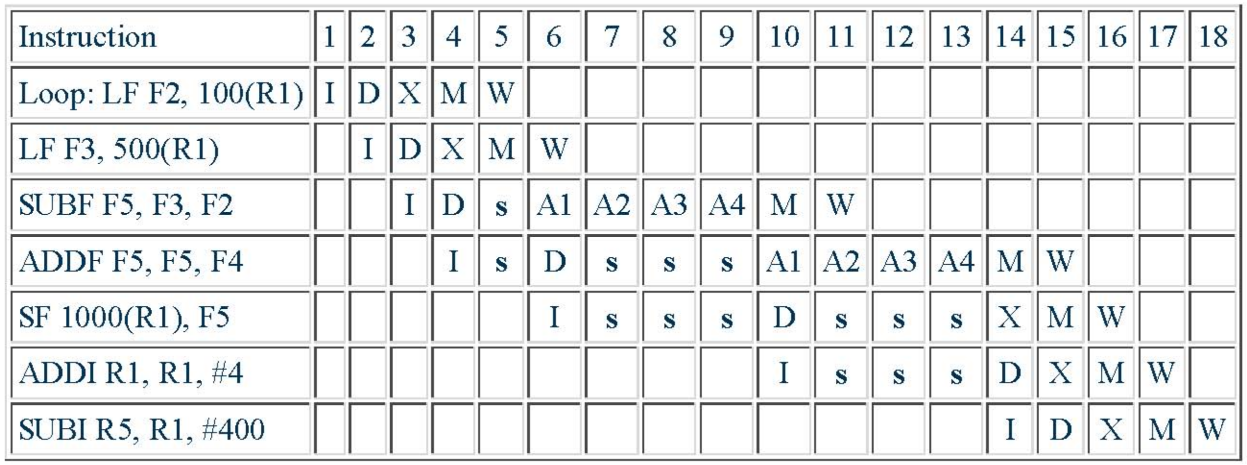
Another FP Example

Model: again, similar to 5 stage pipeline, but with different latencies in the EX stage.

Note: "Latency" means extra delay, or gap, cycles.

Function Unit	Latency
Integer ALU	0
FP ADD/SUB	3
FP MUL	6

```
; sample code
Loop:   LF      F2, 100(R1)
        LF      F3, 500(R1)
        SUBF    F5, F3, F2
        ADDF    F5, F5, F4
        SF      1000(R1), F5
        ADDI    R1, R1, #4
        SUBI    R5, R1, #400
        BNEZ    R5, Loop
```



Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
Loop: LF F2, 100(R1)	I	D	X	M	W													
LF F3, 500(R1)		I	D	X	M	W												
SUBF F5, F3, F2			I	D	s	A1	A2	A3	A4	M	W							
ADDF F5, F5, F4				I	s	D	s	s	s	A1	A2	A3	A4	M	W			
SF 1000(R1), F5						I	s	s	s	D	s	s	s	X	M	W		
ADDI R1, R1, #4										I	s	s	s	D	X	M	W	
SUBI R5, R1, #400														I	D	X	M	W

14 cycles for 7 instructions: CPI = 2

: Cycles per Iteration = 14

hugue

```
;; Unroll 4 times
Loop:
  LF      F2, 100(R1)
  LF      F6, 500(R1)
  LF      F3, 104(R1)
  LF      F7, 504(R1)
  SUBF    F10, F6, F2
  LF      F4, 108(R1)
  SUBF    F11, F7, F3
  LF      F8, 508(R1)
  LF      F5, 112(R1)
  LF      F9, 512(R1)
  SUBF    F12, F8, F4
  SUBF    F13, F9, F5
  ADDI    R1, R1, #16
  ADDF    F10, F10, F14
  ADDF    F11, F11, F14
  ADDF    F12, F12, F14
  ADDF    F13, F13, F14
  SUBI    R5, R1, #400
  SF      984(R1), F10
  SF      988(R1), F11
  SF      992(R1), F12
  SF      996(R1), F13
  BNEZ    R5, Loop
```

27 cycles for 22 instructions: CPI ~ 1.2
Cycles per Iteration = 6.75
Performance > 2x

hugue

Instruction	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16
Loop: LF F2, 100(R1)	I	D	X	M	W											
LF F6, 500(R1)		I	D	X	M	W										
LF F3, 104(R1)			I	D	X	M	W									
LF F7, 504(R1)				I	D	X	M	W								
SUBF F10, F6, F2					I	D	A1	A2	A3	A4	M	W				
LF F4, 108(R1)						I	D	X	M	W						
SUBF F11, F7, F3							I	D	A1	A2	A3	A4	M	W		
LF F8, 508(R1)								I	D	X	s	M	W			
LF F5, 112(R1)									I	D	s	X	s	M	W	
LF F9, 512(R1)										I	s	D	s	X	M	W
SUBF F12, F8, F4												I	s	D	A1	A2
SUBF F13, F9, F5														I	D	A1
ADDI R1, R1, #16															I	D
ADDF F10, F10, F14																I

Instruction	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32
SUBF F12, F8, F4	A3	A4	M	W												
SUBF F13, F9, F5	A2	A3	A4	M	W											
ADDI R1, R1, #16	X	M	W													
ADDF F10, F10, F14	D	A1	A2	A3	A4	M	W									
ADDF F11, F11, F14	I	D	A1	A2	A3	A4	M	W								
ADDF F12, F12, F14		I	D	A1	A2	A3	A4	M	W							
ADDF F13, F13, F14			I	D	A1	A2	A3	A4	M	W						
SUBI R5, R1, #400				I	D	X	s	s	s	M	W					
SF 984(R1), F10					I	D	s	s	s	X	M	W				
SF 988(R1), F11						I	s	s	s	D	X	M	W			
SF 992(R1), F12										I	D	X	M	W		
SF 996(R1), F13											I	D	X	M	W	

A Statically Scheduled *Superscalar* CPU (from P&H 4e)

- Two-issue packets
  - One ALU/branch instruction
  - One load/store instruction
  - 64-bit aligned
    - ALU/branch, then load/store
    - Pad an unused instruction with NOP

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

P&H 4e

Static Multiple Issue and Scheduling

- Compiler groups instructions into “issue packets”
  - Group of instructions that can be issued on a single cycle
  - Determined by pipeline resources required
- Think of an issue packet as a very long instruction
  - Specifies multiple concurrent operations
  - ⇒ Very Long Instruction Word (VLIW)
- Compiler must remove some/all hazards
  - Reorder instructions into issue packets
  - No dependencies within a packet
  - Possibly some dependencies between packets
    - Varies between ISAs; compiler must know!
  - Pad with NOPs if necessary

Address	Instruction type	Pipeline Stages						
n	ALU/branch	IF	ID	EX	MEM	WB		
n + 4	Load/store	IF	ID	EX	MEM	WB		
n + 8	ALU/branch		IF	ID	EX	MEM	WB	
n + 12	Load/store		IF	ID	EX	MEM	WB	
n + 16	ALU/branch			IF	ID	EX	MEM	WB
n + 20	Load/store			IF	ID	EX	MEM	WB

P&H 4e

Hazards in the Dual-Issue MIPS

- More instructions executing in parallel
- EX data hazard
  - Forwarding avoided stalls with single-issue
  - Now can't use ALU result in load/store in same packet

    -
      ```
      add  $t0, $s0, $s1
      load $s2, 0($t0)
      ```
    - Split into two packets, effectively a stall
- Load-use hazard
  - Still one cycle use latency, but now two instructions
- More aggressive scheduling required

P&H 4e

Superscalar Scheduling Example

```
Loop: lw   $t0, 0($s1)       # $t0=array element
      addu $t0, $t0, $s2     # add scalar in $s2
      sw   $t0, 0($s1)       # store result
      addi $s1, $s1,-4       # decrement pointer
      bne  $s1, $zero, Loop # branch $s1!=0
```

	ALU/branch	Load/store	cycle
`Loop:`	`nop`	`lw   $t0, 0($s1)`	1
	`addi $s1, $s1,-4`	`nop`	2
	`addu $t0, $t0, $s2`	`nop`	3
	`bne  $s1, $zero, Loop`	`sw   $t0, 4($s1)`	4

- *IPC = 5/4 = 1.25 (c.f. peak IPC = 2)*

P&H 4e

Loop Unrolling (again)

- Replicate loop body to expose more parallelism
  - Reduces loop-control overhead
- Use different registers per replication
  - Called “register renaming”
  - Avoid loop-carried “anti-dependencies”
    - Store followed by a load of the same register
    - Aka “name dependence”
      - Reuse of a register name

P&H 4e

Loop Unrolling Example

Original

```
Loop: lw    $t0, 0($s1)        # $t0=array element
      addu  $t0, $t0, $s2      # add scalar in $s2
      sw    $t0, 0($s1)        # store result
      addi  $s1, $s1,-4        # decrement pointer
      bne   $s1, $zero, Loop   # branch $s1!=0
```

	ALU/branch	Load/store	cycle
Loop:	nop	lw $t0, 0($s1)	1
	addi $s1, $s1,-4	nop	2
	addu $t0, $t0, $s2	nop	3
	bne $s1, $zero, Loop	sw $t0, 4($s1)	4

	ALU/branch	Load/store	cycle
Loop:	addi $s1, $s1,-16	lw $t0, 0($s1)	1
	nop	lw $t1, 12($s1)	2
	addu $t0, $t0, $s2	lw $t2, 8($s1)	3
	addu $t1, $t1, $s2	lw $t3, 4($s1)	4
	addu $t2, $t2, $s2	sw $t0, 16($s1)	5
	addu $t3, $t4, $s2	sw $t1, 12($s1)	6
	nop	sw $t2, 8($s1)	7
	bne $s1, $zero, Loop	sw $t3, 4($s1)	8

P&H 4e

IPC = 14/8 = 1.75 → Closer to 2, but at cost of registers and code size

How many “unrolls” are needed for there to be no delays?

Back to our previous example -- let’s try 3 unrolls …

	Integer Instruction	FP instruction	cycle
Loop:	`L.D F0,0(R1)`		1
	`L.D F6,-8(R1)`		2
	`L.D F10,-16(R1)`	`ADD.D F4,F0,F2`	3
	`DADDUI R1,R1,#-24`	`ADD.D F8,F6,F2`	4
		`ADD.D F12,F10,F2`	5
	`S.D F4,24(R1)`		6
	`S.D F8,16(R1)`		7
	`BNE R1,R2,Loop`		8
	`S.D F12,8(R1)`		9

```
Loop:
 L.D      F0,0(R1)
 L.D      F6,-8(R1)
 L.D      F10,-16(R1)
 ADD.D    F4,F0,F2
 ADD.D    F8,F6,F2
 ADD.D    F12,F10,F2
 DADDUI   R1,R1,#-24
 S.D      F4,24(R1)
 S.D      F8,16(R1)
 BNE      R1,R2,Loop
 S.D      F12,8(R1)
```

Performance → 9 for 3 unrolls, 3 cycles per loop amortized
Note: “blanks” are NOPs
Note: L.D to ADD.D requires 1 “gap”, ADD.D to S.D requires 2 “gaps”
Note: L/S offsets must be adjusted as L.D/S.D get rescheduled

H&P 3e

How many “unrolls” are needed for there to be no delays?

Now let’s try 4 unrolls …

	Integer Instruction	FP instruction	cycle
Loop:	`L.D F0,0(R1)`		1
	`L.D F6,-8(R1)`		2
	`L.D F10,-16(R1)`	`ADD.D F4,F0,F2`	3
	`L.D F14,-24(R1)`	`ADD.D F8,F6,F2`	4
	`DADDUI R1,R1,#-32`	`ADD.D F12,F10,F2`	5
	`S.D F4,32(R1)`	`ADD.D F16,F14,F2`	6
	`S.D F8,24(R1)`		7
	`S.D F12,16(R1)`		8
	`BNE R1,R2,Loop`		9
	`S.D F16,8(R1)`		10

```
Loop:
 L.D        F0,0(R1)
 L.D        F6,-8(R1)
 L.D        F10,-16(R1)
 L.D        F14,-24(R1)
 ADD.D      F4,F0,F2
 ADD.D      F8,F6,F2
 ADD.D      F12,F10,F2
 ADD.D      F16,F14,F2
 DADDUI     R1,R1,#-32
 S.D        F4,32(R1)
 S.D        F8,24(R1)
 S.D        F12,16(R1)
 BNE        R1,R2,Loop
 S.D        F16,8(R1)
```

Performance → 10 for 4 unrolls, 2.5 cycles per loop amortized
Note: Many FP slots are empty. FP utilization is 4/10 or 40%. Can improve this slightly with more unrolling.

H&P 3e

Back to Matrix Multiply: Loop Unrolling

```
// Starting point -- blocked code
for (i=0; i<N; i=i+B)
  for (j=0; j<N; j=j+B)
    for (k=0; k<N; k=k+B)
      for (ii=i; ii<i+B; ii=ii+1)
        for (jj=j; jj<j+B; jj=jj+1)
          for (kk=k; kk<k+B; kk=kk+1)
            x[ii][jj] += y[ii][kk] * z[kk][jj];
```

Work on basic block furthest inside loop ...

```
// unroll the loop 8 times
for (i=0; i<N; i=i+B)
  for (j=0; j<N; j=j+B)
    for (k=0; k<N; k=k+B)
      for (ii=i; ii<i+B; ii=ii+1)
        for (jj=j; jj<j+B; jj=jj+1)
          for (kk=k; kk<k+B; kk=kk+8) {
            x[ii][jj] += y[ii][kk] * z[kk][jj];
            x[ii][jj] += y[ii][kk+1] * z[kk+1][jj];
            x[ii][jj] += y[ii][kk+2] * z[kk+2][jj];
            x[ii][jj] += y[ii][kk+3] * z[kk+3][jj];
            x[ii][jj] += y[ii][kk+4] * z[kk+4][jj];
            x[ii][jj] += y[ii][kk+5] * z[kk+5][jj];
            x[ii][jj] += y[ii][kk+6] * z[kk+6][jj];
            x[ii][jj] += y[ii][kk+7] * z[kk+7][jj];
          }
```

Matrix Multiply: Scalar Replacement

Scalar Replacement
Enables replacement of variables with registers.

Scalar replacement can also interfere with vectorization

```
// Scalar Replacement
for (i=0; i<N; i=i+NB)
  for (j=0; j<N; j=j+NB)
    for (k=0; k<N; k=k+NB)
      for (ii=i; ii<i+NB; ii=ii+1)
        for (jj=j; jj<j+NB; jj=jj+1)
          for (kk=k; kk<k+NB; kk=kk+4) {
            t0 = x[ii][jj];
            t1 = y[ii][kk];
            t2 = z[kk][jj];
            t3 = y[ii][kk+1];
            t4 = z[kk+1][jj];
            t5 = y[ii][kk+2];
            t6 = z[kk+2][jj];
            t7 = y[ii][kk+3];
            t8 = z[kk+3][jj];
            t0 = t0 + t1 x t2;
            t0 = t0 + t3 x t4;
            t0 = t0 + t5 x t6;
            t0 = t0 + t7 x t8;
            x[ii][jj] = t0;
          }
```

Matrix Multiply: Scalar Replacement

Scalar Replacement
Enables replacement of variables with registers.

Now, break up operations into simpler components.

```
// Scalar Replacement and break up operations.
for (i=0; i<N; i=i+B)
  for (j=0; j<N; j=j+B)
    for (k=0; k<N; k=k+B)
      for (ii=i; ii<i+B; ii=ii+1)
        for (jj=j; jj<j+B; jj=jj+1)
          for (kk=k; kk<k+B; kk=kk+4) {
            t0 = x[ii][jj];
            t1 = y[ii][kk];
            t2 = z[kk][jj];
            t3 = y[ii][kk+1];
            t4 = z[kk+1][jj];
            t5 = y[ii][kk+2];
            t6 = z[kk+2][jj];
            t7 = y[ii][kk+3];
            t8 = z[kk+3][jj];
            t1 = t1 x t2;
            t0 = t0 + t1;
            t3 = t3 x t4;
            t0 = t0 + t3;
            t5 = t5 x t6;
            t0 = t0 + t5;
            t7 = t7 x t8;
            t0 = t0 + t7;
            x[ii][jj] = t0;
          }
```

Matrix Multiply: Skewing

Skewing
Reorder operations taking latencies into account.

Ex: Let multiply take two cycles

Alternative →
Software pipelining!
(move later operations to previous iterations)

```
// Scalar Replacement and break up operations.
for (i=0; i<N; i=i+B)
  for (j=0; j<N; j=j+B)
    for (k=0; k<N; k=k+B)
      for (ii=i; ii<i+B; ii=ii+1)
        for (jj=j; jj<j+B; jj=jj+1)
          for (kk=k; kk<k+B; kk=kk+4) {
            t0 = x[ii][jj];
            t1 = y[ii][kk];
            t2 = z[kk][jj];
            t3 = y[ii][kk+1];
            t4 = z[kk+1][jj];
            t5 = y[ii][kk+2];
            t6 = z[kk+2][jj];
            t7 = y[ii][kk+3];
            t8 = z[kk+3][jj];
            t1 = t1 x t2;
            t3 = t3 x t4;
            t0 = t0 + t1;
            t5 = t5 x t6;
            t0 = t0 + t3;
            t7 = t7 x t8;
            t0 = t0 + t5;
            t0 = t0 + t7;
            x[ii][jj] = t0;
          }
```

Matrix Multiply: "Micro" Blocking

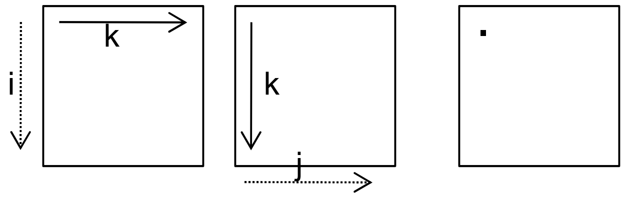
Multilevel blocking with the idea that the innermost level is for register mapping.
Note: innermost loop has change index order. Now ***KIJ****.*

```
// Two levels of blocking.
// Mini-MMM has single blocking factor NB.
// Micro-MMM has a different one for each dimension:  UIB, UJB, UKB
for (i=0; i<N; i=i+NB)
  for (j=0; j<N; j=j+NB)
    for (k=0; k<N; k=k+NB)
      for (ii=i; ii<i+NB; ii=ii+UIB)                // Mu
        for (jj=j; jj<j+NB; jj=jj+UJB)              // Nu
          for (kk=k; kk<k+NB; kk=kk+UKB)            // Ku
            for (kkk = kk; kkk<kk+UKB; kkk=kkk+1)
              for (iii = ii; iii<ii+UIB; iii=iii+1)
                for (jjj = jj; jjj<jj+UJB; jjj=jjj+1)
                  x[iii][jjj] += y[iii][kkk] * z[kkk][jjj];
```

Matrix Multiply: "Micro" Blocking

Multilevel blocking with the idea that the innermost level is for register mapping. Now K as outermost loop for instruction parallelism:

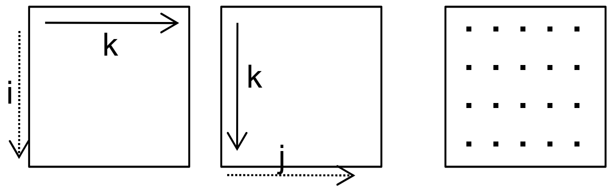
loop order = ijk



Block across k. Then single output which becomes bottleneck accumulator.

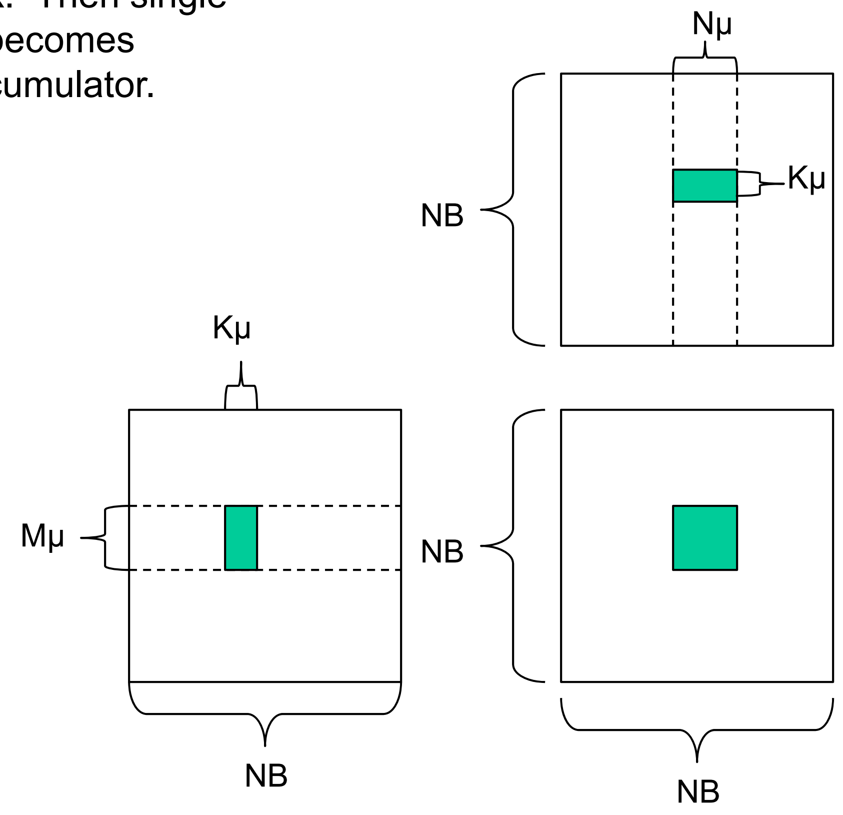
2n instructions
-- n parallel mults
-- n dependent adds
→ why dependent? See accumulator problem below
(even with reduction logN steps)
but: 2n+1 live variables

loop order = kij



Block across i and j. Then N^2 outputs which become independent accumulators.

$2n^2$ instructions
-- n^2 parallel mults
-- n^2 parallel adds
but: n^2 + n live variables



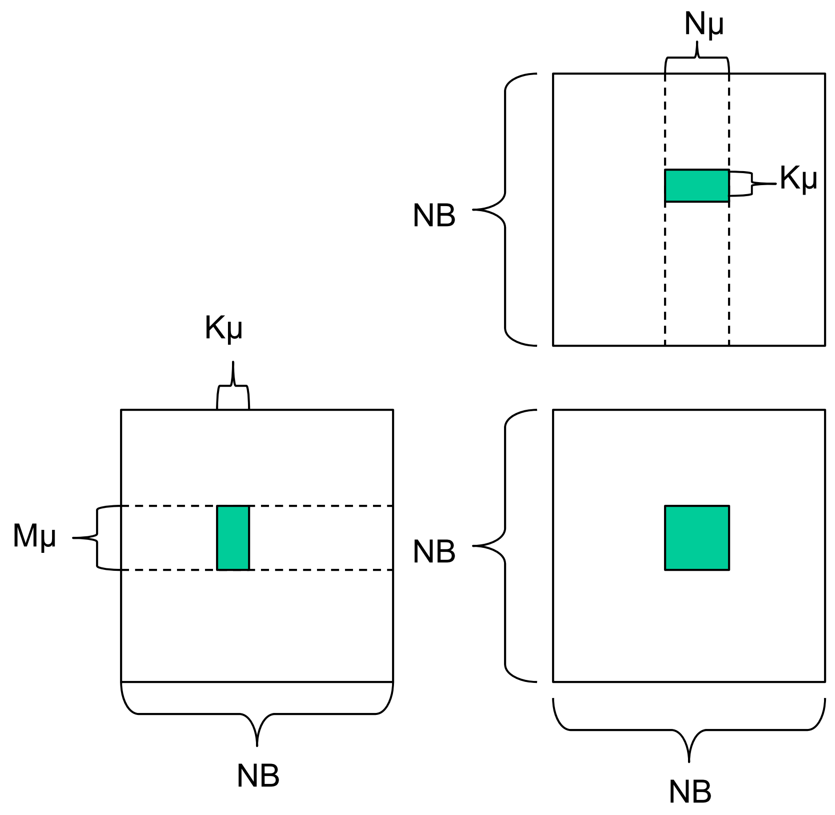
C&F&P

Sizing the micro loops

Mu * Nu + Mu + Nu ≤ # of registers

Let Mu = Nu = 2, Let Ku = 1

(looking at two nested loops)



C&F&P

The Accumulator Problem

```
X = X + Y[0]
X = X + Y[1]
X = X + Y[2]
X = X + Y[3]

L.D     F1,(R1)
ADD.D   F0,F0,F1
L.D     F1,8(R1)
ADD.D   F0,F0,F1
L.D     F1,16(R1)
ADD.D   F0,F0,F1
L.D     F1,24(R1)
ADD.D   F0,F0,F1
S.D     F0,(R2)
```

All latencies = 1

	Integer Instruction	FP instruction	cycle
Loop:	`L.D F1,0(R1)`		1
	`L.D F1,8(R1)`	`ADD.D F0,F0,F1`	2
	`L.D F1,16(R1)`	`ADD.D F0,F0,F1`	3
	`L.D F1,24(R1)`	`ADD.D F0,F0,F1`	4
		`ADD.D F0,F0,F1`	5
	`S.D F0,(R2)`		6
			7
			8
			9
			10
			11

Unrolling and SW pipelining gets nearly full utilization

The Accumulator Problem

```
X = X + Y[0]
X = X + Y[1]
X = X + Y[2]
X = X + Y[3]

L.D     F1,(R1)
ADD.D   F0,F0,F1
L.D     F1,8(R1)
ADD.D   F0,F0,F1
L.D     F1,16(R1)
ADD.D   F0,F0,F1
L.D     F1,24(R1)
ADD.D   F0,F0,F1
S.D     F0,(R2)
```

Let L.D latency = 3

	Integer Instruction	FP instruction	cycle
Loop:	L.D F1,0(R1)		1
	L.D F2,8(R1)		2
	L.D F3,16(R1)		3
	L.D F4,24(R1)	ADD.D F0,F0,F1	4
		ADD.D F0,F0,F2	5
		ADD.D F0,F0,F3	6
		ADD.D F0,F0,F4	7
	S.D F0,(R2)		8
			9
			10
			11

Unrolling and SW pipelining still gets nearly full utilization

```
X = X + Y[0]
X = X + Y[1]
X = X + Y[2]
X = X + Y[3]

L.D    F1,(R1)
ADD.D  F0,F0,F1
L.D    F1,8(R1)
ADD.D  F0,F0,F1
L.D    F1,16(R1)
ADD.D  F0,F0,F1
L.D    F1,24(R1)
ADD.D  F0,F0,F1
S.D    F0,(R2)
```

Let L.D latency = 1, ADD.D latency = 3

	Integer Instruction	FP instruction	cycle
Loop:	L.D F1,0(R1)		1
		ADD.D F0,F0,F1	2
			3
	L.D F1,8(R1)		4
		ADD.D F0,F0,F1	5
			6
	L.D F1,16(R1)		7
		ADD.D F0,F0,F1	8
			9
	L.D F1,24(R1)		10
		ADD.D F0,F0,F1	11
			12
			13
	S.D F0,(R2)		14

Problem: **RAW** dependency on F0

Note: the **WAW** dependency is NOT (likely to be) a serious problem

Solutions to accumulator problem

1. Reduction Tree (on board)
2. Multiple accumulators (special case of reduction tree)

```
X0 = Y[0]
X1 = Y[1]
X2 = Y[2]
X3 = Y[3]
for (i = 4; i < COUNT; i = i+4) {
  X0 = X0 + Y[i+0];
  X1 = X1 + Y[i+1];
  X2 = X2 + Y[i+2];
  X3 = X3 + Y[i+3];
}
X0 = X0 + X1;
X2 = X2 + X3;
X = X2 + X4;
```

```
Loop:
  L.D     F4,(R1)
  ADD.D   F0,F0,F4
  L.D     F5,8(R1)
  ADD.D   F1,F1,F5
  L.D     F6,16(R1)
  ADD.D   F2,F2,F6
  L.D     F7,24(R1)
  ADD.D   F3,F3,F7
```

```
X0 = X0 + Y[i+0];
X1 = X1 + Y[i+1];
X2 = X2 + Y[i+2];
X3 = X3 + Y[i+3];
```

Let L.D latency = 1, ADD.D latency = 3

	Integer Instruction	FP instruction	cycle
Loop:	L.D F4,0(R1)		1
	L.D F5,8(R1)	ADD.D F0,F0,F4	2
	L.D F6,16(R1)	ADD.D F1,F1,F5	3
	L.D F7,24(R1)	ADD.D F2,F2,F6	4
	L.D F4,32(R1)	ADD.D F3,F3,F7	5
	L.D F5,40(R1)	ADD.D F0,F0,F4	6
	L.D F6,48(R1)	ADD.D F1,F1,F5	7
	L.D F7,56(R1)	ADD.D F2,F2,F6	8
		ADD.D F3,F3,F7	9
			10
			11
			12

```
 L.D     F4,(R1)
 ADD.D   F0,F0,F4
 L.D     F5,8(R1)
 ADD.D   F1,F1,F5
 L.D     F6,16(R1)
 ADD.D   F2,F2,F6
 L.D     F7,24(R1)
 ADD.D   F3,F3,F7
.
.
.
```

Another Example: Add one vector to another, element by element

```
X[0] = X[0] + Y[0]
X[1] = X[1] + Y[1]
X[2] = X[2] + Y[2]
X[3] = X[3] + Y[3]
```

```
L.D     F0,(R0)
L.D     F1,(R1)
ADD.D   F0,F0,F1
S.D     F0,(R2)
L.D     F0,8(R0)
L.D     F1,8(R1)
ADD.D   F2,F0,F1
S.D     F2,8(R2)
L.D     F0,16(R0)
L.D     F1,16(R1)
ADD.D   F3,F0,F1
S.D     F3,16(R2)
L.D     F1,24(R0)
L.D     F0,24(R1)
ADD.D   F4,F0,F1
S.D     F4,24(R2)
```

Let L.D latency = 1, ADD.D latency = 3

	Integer Instruction	FP instruction	cycle
Loop:	`L.D F0,0(R0)`		1
	`L.D F1,0(R1)`		2
	`L.D F0,8(R0)`	`ADD.D F0,F0,F1`	3
	`L.D F1,8(R1)`		4
	`L.D F0,16(R0)`	`ADD.D F2,F0,F1`	5
	`L.D F1,16(R1)`		6
	`L.D F0,24(R0)`	`ADD.D F3,F0,F1`	7
	`L.D F1,24(R1)`		8
	`S.D F0,(R2)`	`ADD.D F4,F0,F1`	9
	`S.D F2,8(R2)`		10
	`S.D F3,16(R2)`		11
	`S.D F4,24(R2)`		12

MMM -- redo scalar replacement

```
// Scalar Replacement and break up
// operations.
for (i=0; i<N; i=i+1)
  for (j=0; j<N; j=j+1)
    for (k=0; k<N; k=k+4) {
//     x[i][j] += y[i][k] * z[k][j];
//     x[i][j] += y[i][k+1] * z[k+1][j];
//     x[i][j] += y[i][k+2] * z[k+2][j];
//     x[i][j] += y[i][k+3] * z[k+3][j];
       t0 = x[i][j];
       t1 = y[i][k];
       t2 = z[k][j];
       t3 = y[i][k+1];
       t4 = z[k+1][j];
       t5 = y[i][k+2];
       t6 = z[k+2][j];
       t7 = y[ii][kk+3];
       t8 = z[kk+3][jj];
       t1 = t1 x t2;
       t3 = t3 x t4;
       t0 = t0 + t1;
       t5 = t5 x t6;
       t0 = t0 + t3;
       t7 = t7 x t8;
       t0 = t0 + t5;
       t0 = t0 + t7;
       x[i][j] = t0;
      }
```

```
// Scalar Replacement and break up
// operations.
for (k=0; k<N; k=k+1)
  for (j=0; j<N; j=j+Mu)       // Mu = 2
    for (i=0; i<N; i=k+Nu) {   // Nu = 2
//     x[i][j] += y[i][k] * z[k][j];
//     x[i+1][j] += y[i+1][k] * z[k][j];
//     x[i][j+1] += y[i][k] * z[k][j+1];
//     x[i+1][j+1] += y[i+1][k] * z[k][j+1];
       t0 = x[i][j];
       t1 = x[i+1][j];
       t2 = x[i][j+1];
       t3 = x[i+1][j+1];
       t4 = y[i][k];
       t5 = z[k][j];
       t6 = y[i+1][k];
       t7 = z[k][j+1];
       t8 = t4 * t5;    // t4 & t5 reused
       t0 = t0 + t8;
       x[i][j] = t0;
       t5 = t5 * t6;    // done w/ t5
       t1 = t1 + t5;
       x[i+1][j] = t1;
       t4 = t4 * t7;    // done w/ t4
       t2 = t2 + t4;
       x[i][j+1] = t2;
       t6 = t6 * t7;
       t3 = t3 + t6;
       x[i+1][j+1] = t3;       }
```

More variations for MMM

1. Skewing -- delay multiplies by appropriate number of cycles (as previously)
   - this works for simple pipelines
   - works for pipelines with single FP issue per cycle
   - See assignment 3

2. Assume separate FMUL and FADD pipelines
   - this is more like contemporary high-end processors
   - now load and store are clearly the bottleneck

```
// Scalar Replacement and break up
// operations.
for (k=0; k<N; k=k+1)
  for (j=0; j<N; j=j+Mu)       // Mu = 2
    for (i=0; i<N; i=k+Nu) {   // Nu = 2
//     x[i][j] += y[i][k] * z[k][j];
//     x[i+1][j] += y[i+1][k] * z[k][j];
//     x[i][j+1] += y[i][k] * z[k][j+1];
//     x[i+1][j+1] += y[i+1][k] * z[k][j+1];
       t0 = x[i][j];
       t1 = x[i+1][j];
       t2 = x[i][j+1];
       t3 = x[i+1][j+1];
       t4 = y[i][k];
       t5 = z[k][j];
       t6 = y[i+1][k];
       t7 = z[k][j+1];
       t8 = t4 * t5;    // t4 & t5 reused
       t0 = t0 + t8;
       x[i][j] = t0;
       t5 = t5 * t6;    // done w/ t5
       t1 = t1 + t5;
       x[i+1][j] = t1;
       t4 = t4 * t7;    // done w/ t4
       t2 = t2 + t4;
       x[i][j+1] = t2;
       t6 = t6 * t7;
       t3 = t3 + t6;
       x[i+1][j+1] = t3;        }
```

```
// *X = R0 *Y = R1 *Z = R2
Loop:
  L.D    F4,(R1)           ; Y(i,k)
  L.D    F6,(R2)           ; Z(k,j)
  L.D    F5,ROWSIZE(R2)    ; Y(i+1,k)
  L.D    F7,8(R2)          ; Z(k,j+1)
  L.D    F0,(R0)           ; X(i,j)
  L.D    F2,8(R0)          ; X(i,j+1)
  L.D    F1,ROWSIZE(R0)    ; X(i+1,j)
  L.D    F3,ROWSIZE+8(R0); X(i+1,j+1)
  MUL.D  F8,F4,F6          ; y[i][k]*z[k][j]
  MUL.D  F4,F4,F7          ; y[i][k]*z[k][j+1]
  MUL.D  F6,F5,F6          ; y[i+1]*z[k][j]
  MUL.D  F7,F5,F7          ; y[i+1][k]*z[k][j+1]
  ADD.D  F0,F0,F8          ; X[i][j]
  ADD.D  F2,F2,F4          ; X[i][j+1]
  ADD.D  F1,F1,F6          ; X[i+1][j]
  ADD.D  F3,F3,F7          ; X[i+1][j+1]
  S.D    F0,(R0)
  S.D    F2,8(R0)
  S.D    F1,ROWSIZE(R0)
  S.D    F3,ROWSIZE+8(R0)
```

Let → L.D latency = 1, ADD.D latency = MUL.D latency = 3

	Integer Instruction	FPADD	FPMUL	cycle
Loop:	**L.D F4,(R1)**			1
	L.D F6,(R2)			2
	L.D F5,ROWSIZE(R2)		**MUL.D F8,F4,F6**	3
	L.D F7,8(R2)		**MUL.D F4,F4,F7**	4
	L.D F0,(R0)		**MUL.D F6,F5,F6**	5
	L.D F2,8(R0)	**ADD.D F0,F0,F8**	**MUL.D F7,F5,F7**	6
	L.D F1,ROWSIZE(R0)	**ADD.D F2,F2,F4**		7
	L.D F3,ROWSIZE+8(R0)	**ADD.D F1,F1,F6**		8
	S.D F0,(R0)	**ADD.D F3,F3,F7**		9
	S.D F2,8(R0)			10
	S.D F1,ROWSIZE(R0)			11
	S.D F3,ROWSIZE+8(R0)			12

Software Pipelining

<u>**Software Pipelining**</u> ⇔ A technique for reorganizing loops such that each iteration is made from instructions chosen from different iterations of the original loop.

- *By choosing instructions from different iterations, dependent computations are separated from one another by an entire loop body, increasing the possibility that the unrolled loop can be scheduled without stalls.*

H&P 3e

Software Pipelining: Same Example

```
for (i = 1000; i > 0; i=i-1)
  x[i] = x[i] + s;

; Original Code
Loop:   L.D     F0,0(R1)
        ADD.D   F4,F0,F2
        S.D     F4,0(R1)
        DADDUI  R1,R1,#-8
        BNE     R1,R2,Loop
```

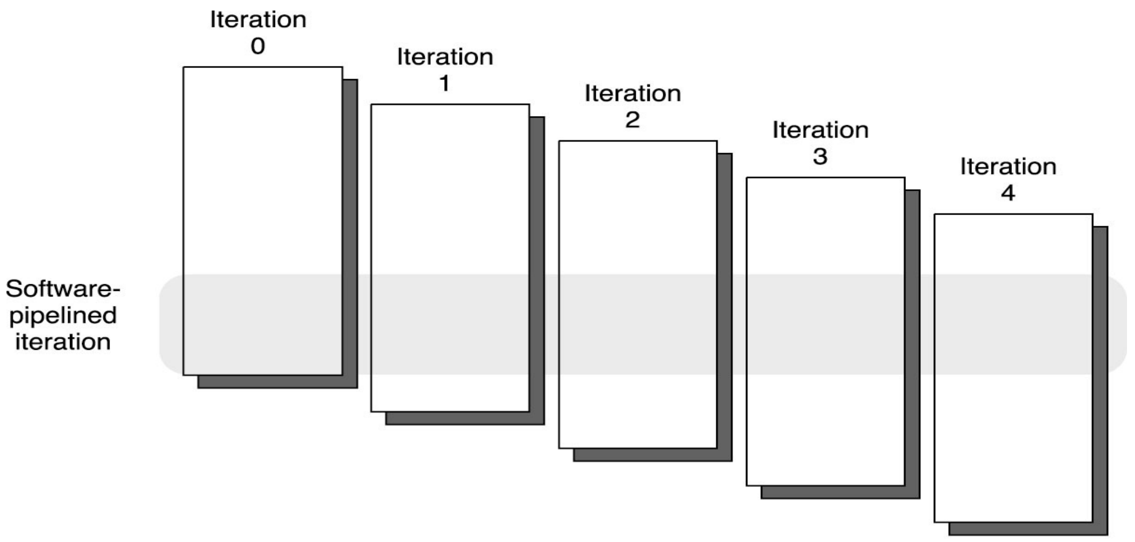
```
                                Clock Cycle Issued
Loop:   L.D     F0,0(R1)                1
        stall                           2
        ADD.D   F4,F0,F2                3
        stall                           4
        stall                           5
        S.D     F4,0(R1)                6
        DADDUI  R1,R1,#-8               7
        stall                           8
        BNE     R1,R2,Loop              9
        stall                           10

;; 10 cycles per iteration, CPI = 2
```

```
; Show iterations
;  w/o loop overhead
iter 1: L.D     F0,0(R1) //128
        ADD.D   F4,F0,F2
        S.D     F4,0(R1) //128
iter 2: L.D     F0,0(R1) //120
        ADD.D   F4,F0,F2
        S.D     F4,0(R1) //120
iter 3: L.D     F0,0(R1) //112
        ADD.D   F4,F0,F2
        S.D     F4,0(R1) //112
```

```
; SW Pipelined Code – count backwards
;   Loop only           Ex:  R1 = 112
Loop:   S.D     F4,16(R1) ; store M[i]
        ADD.D   F4,F0,F2  ; add to M[i-1]
        L.D     F0,0(R1)  ; load M[i-2]
        DADDUI  R1,R1,#-8
        BNE     R1,R2,Loop
```

H&P 3e

Software Pipelining: Procedure



```
; 1st iteration R1 = 128
iter 1: L.D       F0,0(R1)
        ADD.D     F4,F0,F2
        S.D       F4,0(R1)
```

```
; 2nd iteration R1 = 120
iter 2: L.D       F0,0(R1)
        ADD.D     F4,F0,F2
        S.D       F4,0(R1)
```

```
; 3rd iteration R1 = 112
iter 3: L.D       F0,0(R1)
        ADD.D     F4,F0,F2
        S.D       F4,0(R1)
```

```
for (i = 1000; i > 0; i=i-1)
  x[i] = x[i] + s;

; Original Code
Loop:   L.D       F0,0(R1)
        ADD.D     F4,F0,F2
        S.D       F4,0(R1)
        DADDUI    R1,R1,#-8
        BNE       R1,R2,Loop
```

```
; SW Pipelined Code
;   Loop only
Loop:   S.D       F4,16(R1)  ; store M[i]
        ADD.D     F4,F0,F2   ; add to M[i-1]
        L.D       F0,0(R1)   ; load M[i-2]
        DADDUI    R1,R1,#-8
        BNE       R1,R2,Loop
```

H&P 3e

Software Pipelining: Timing

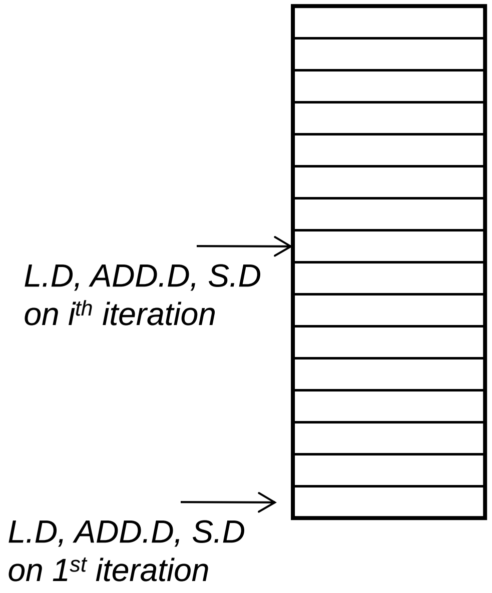
```
for (i = 1000; i > 0; i=i-1)
  x[i] = x[i] + s;

; Original Code
Loop:    L.D      F0,0(R1)
         ADD.D    F4,F0,F2
         S.D      F4,0(R1)
         DADDUI   R1,R1,#-8
         BNE      R1,R2,Loop
```

```
                                  Clock Cycle Issued
Loop:      L.D        F0,0(R1)              1
           stall                            2
           ADD.D      F4,F0,F2              3
           stall                            4
           stall                            5
           S.D        F4,0(R1)              6
           DADDUI     R1,R1,#-8 7
           stall                            8
           BNE        R1,R2,Loop            9
           stall                            10
```

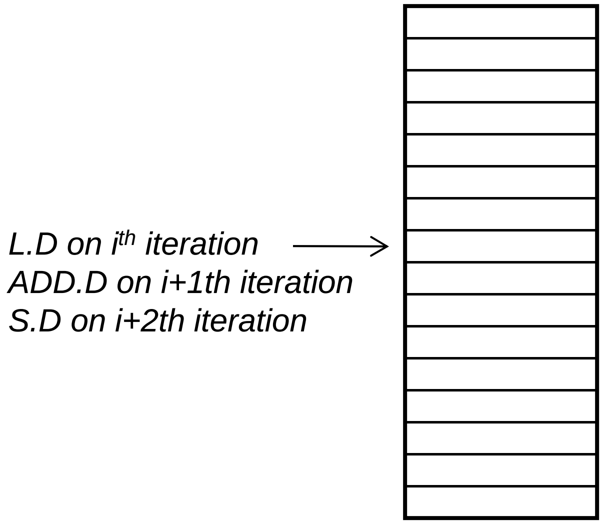
```
; SW Pipelined Code
;   Loop only – not yet scheduled            clock cycle issued
Loop:   S.D     F4,16(R1) ; store M[i]                1
        ADD.D   F4,F0,F2  ; add to M[i-1]             2
        L.D     F0,0(R1)  ; load M[i-2]               3
        DADDUI  R1,R1,#-8                             4
        stall                                         5
        BNE     R1,R2,Loop                            6
        stall                                         7

;; 7 cycles per iteration, CPI = 1.4
```

H&P 3e

```
for (i = 1000; i > 0; i=i-1)
  x[i] = x[i] + s;

; Original Code  R1 = 64
Loop:   L.D     F0,0(R1)   ; load x[i]
        ADD.D   F4,F0,F2   ; add to x[i]
        S.D     F4,0(R1)   ; store x[i]
        DADDUI  R1,R1,#-8
        BNE     R1,R2,Loop
```

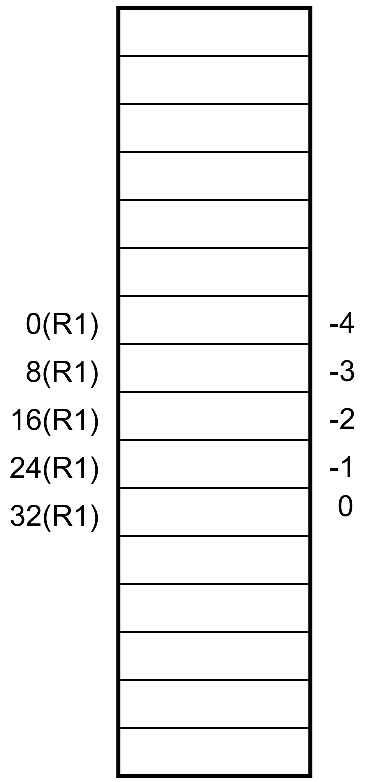

L.D, ADD.D, S.D on ith iteration
L.D, ADD.D, S.D on 1st iteration

```
; SW Pipelined Code  R1 = 64
Loop:   S.D     F4,16(R1) ; store x[i]    80
        ADD.D   F4,F0,F2  ; add to x[i-1] 72
        L.D     F0,0(R1)  ; load x[i-2]   62
        DADDUI  R1,R1,#-8
        BNE     R1,R2,Loop
```



H&P 3e

```
; How this works …  R1 = 64
; Unroll SW Pipelined Code 3x

Loop:  S.D     F4,32(R1) ; store  x[i]   96
       ADD.D   F4,F0,F2  ; add to x[i-1] 88
       L.D     F0,16(R1) ; load   x[i-2] 80
       S.D     F4,24(R1) ; store  x[i-1] 88
       ADD.D   F4,F0,F2  ; add to x[i-2] 80
       L.D     F0,8(R1)  ; load   x[i-3] 72
       S.D     F4,16(R1) ; store  x[i-2] 80
       ADD.D   F4,F0,F2  ; add to x[i-3] 72
       L.D     F0,0(R1)  ; load   x[i-4] 64
       DADDUI  R1,R1,#-24
       BNE     R1,R2,Loop
```



Unrolling only done to demonstrate execution. Assume that code can be scheduled to eliminate stalls, unrolling does not improve performance here.

Why? With a superscalar CPU and branch prediction, overhead instructions are completely hidden.

H&P 3e

Software Pipelining: Requires new start-up and tear-down code

```
; Original Code
Loop:    L.D      F0,0(R1)
         ADD.D    F4,F0,F2
         S.D      F4,0(R1)
         DADDUI   R1,R1,#-8
         BNE      R1,R2,Loop
```

```
; SW Pipelined Code -- with start-up and tear-down code
;                                                    clock cycle issued
        L.D     F0,0(R1)   ; 1st iteration
        stall
        ADD.D   F4,F0,F2   ; 1st iteration
        L.D     F0,-8(R1)  ; 2nd iteration
        DSUBUI  R1,R1,#16  ; start on "3rd" iteration
Loop:   S.D     F4,16(R1)  ; store M[i]                 1
        ADD.D   F4,F0,F2   ; add to M[i-1]              2
        L.D     F0,0(R1)   ; load M[i-2]                3
        DADDUI  R1,R1,#-8                               4
        stall                                           5
        BNE     R1,R2,Loop                              6
        stall                                           7
        S.D     16(R1),F4
        ADD.D   F4,F0,F2
        stall
        stall
        S.D     8(R1),F4
```

H&P 3e

Software Pipelining: Get rid of remaining stalls

```
; Clean up remaining stalls, account for small # of iterations
;                                                   clock cycle issued
       DSUBUI  R1,R1,#8       ; initialize R1 and F2
       L.D     F0,8(R1)       ; 1st iteration -- load M[n]
       BEZ     R1,one_iter    ;
       ADD.D   F4,F0,F2       ; 1st iteration -- add to M[n]
       DSUBUI  R1,R1,#8
       L.D     F0,8(R1)       ; 2nd iteration -- load M[n-1]
       BEZ     R1,two_iters
Loop:  S.D     F4,16(R1)      ; store M[i]
       DADDUI  R1,R1,#-8
       ADD.D   F4,F0,F2       ; add to M[i-1]
       BNEZ    R1,Loop
       L.D     F0,0(R1)       ; load M[i-2] -- delay slot
; out of loop
       S.D     16(R1),F4      ; store M[2]
two_iters:
       ADD.D   F4,F0,F2       ; add to M[1]
one_iter:
       stall
       stall
       S.D     8(R1),F4       ; store M[1]
```

H&P 3e

SW Pipelining Comments

→ some case are much harder, e.g., if a register must be saved over several iterations. This makes start-up and tear-down much longer.

→ But then SW Pipelining can be combined with loop unrolling.

SW Pipelining vs. Loop Unrolling

→ SW Pipelining requires less code

→ Loop Unrolling reduces overhead per loop

→ SW Pipelining improves efficiency

SW Pipelining vs. Loop Unrolling + Code Scheduling

→ SW Pipelining is a systematic way of Code Scheduling

H&P 3e