UNIT 2

5/2/24

Advanced Cache Optimization Techniques

* REDUCING HIT TIME → Small and simple first-level caches and way-prediction (generally ↓ power consumption)

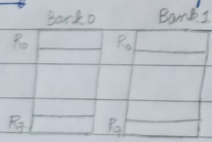
* INCREASING CACHE BANDWIDTH → Pipelined caches
  → Multibanked caches
  → Nonblocking caches
  } varying impact on power consumption

* REDUCING THE MISS PENALTY → Critical word first and merging write buffers (little impact on power)

* REDUCING MISS RATE → Compiler optimizations (improves power consumption)

* REDUCING MISS PENALTY/MISS RATE VIA PARALLELISM
  → hardware prefetching and compiler prefetching
  (↑ power consumption ∵ prefetched data unused)

Banking → There are temporary registers defined internally



You can switch between banks
Each bank is independent
4 banks ideal

Cache
- Blocking: Sequential execution. Processor made to stall in case of miss. In-order execution (default)
- Non-blocking: Out-of-order execution. Instructions having no dependencies are executed first → independent instructions

6/2/24

(1) SMALL AND SIMPLE FIRST-LEVEL CACHES

Cache-hit critical path, 3 steps:
⇒ addressing the tag memory using the index portion of the address
⇒ comparing the read tag value to the address
⇒ setting the multiplexor to choose the correct data item if the cache is set associative.

* L2 cache small enough to fit on chip with processor

Example Appendix B
Determine whether a 32KB 4-way set associative L1 cache has a faster memory access time than a 32KB 2-way set associative L1 cache. Assume the miss penalty to L2 is 15 times the access time for the faster L1 cache. Ignore misses beyond L2. Which has the faster average memory access time?

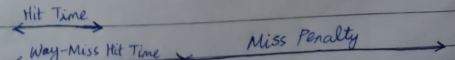
$AMAT_{2\text{-way}} = 1 + 0.038 \times 15 = 1.38$ (1 by default, unless otherwise specified)

$AMAT_{4\text{-way}} = 1 \times 1.4 + 0.037 \times 10 = 1.77$ (15/1.4 = 10.1 ≃ 10)

∵ 4-way cache, the access time is 1.4 times longer

(2) FAST HIT TIMES VIA WAY PREDICTION

WAY PREDICTION: Keep extra bits in cache to predict "way" (block within set) of next access
→ Multiplexer set early to select desired block; only 1 tag comparison done that cycle (in parallel with reading data)
→ Miss ⇒ check other blocks for matches in next cycle
* Accuracy ≃ 85%



DRAWBACK: CPU pipeline harder if hit time is variable-length

(3) INCREASING CACHE BANDWIDTH BY PIPELINING IN

Simply to pipeline cache access
→ Multiple clock cycle for L1 cache hit

ADVANTAGE: Fast cycle time and slow hit
DRAWBACK: Increasing the number of pipeline stages leads to
→ greater penalty on mispredicted branches
→ more clock cycles between the issue of load and use of the data

(4) NON-BLOCKING CACHES TO INCREASE CACHE BANDWIDTH
→ LOCK-UP FREE CACHE allows continued cache hits during miss

Hit under miss → reduces effective miss penalty by working during miss vs. ignoring CPU requests

Hit under multiple miss / miss under miss
→ further lowers effective miss penalty by overlapping multiple misses

Example: Which is more important for floating point programs: 2-way set associativity or hit under one miss for the primary data caches? What about integer programs? Assume the following average miss rates for 32KB data caches: 5.2% for floating-point programs with a direct mapped cache, 4.9% for these programs with a 2-way set associative cache, 3.5% for integer programs with a direct-mapped cache, and 3.2% for integer programs with a 2-way set associative cache. Assume miss penalty to L2 is 10 cycles & the L2 misses & penalties are the same.

FLOATING-POINT PROGRAMS

AMAT: Miss rate$_{DM}$ × Miss penalty = 5.2% × 10 = 0.52
AMAT: Miss rate$_{2\text{-way}}$ × Miss penalty = 4.9% × 10 = 0.49

$\frac{1}{0.52} = 0.9423$
$\frac{0.49}{0.52} = 94.23\%$

INTEGER PROGRAMS

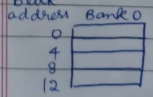
AMAT: Miss rate$_{DM}$ × Miss penalty = 3.5% × 10 = 0.35
AMAT: Miss rate$_{2\text{-way}}$ × Miss penalty = 3.2% × 10 = 0.32

$\frac{0.32}{0.35} = 0.914$
$\frac{0.35}{0.32} = 91.4\%$

(5) Sequential Interleaving

- Works best when accesses naturally spread across banks
  ⇒ mapping of addresses to banks affects behaviour of memory system
- Simple mapping that works well is sequential interleaving
  → Spread block addresses sequentially across banks

Slack Time
Switch b/w processes : pseudo parallelism



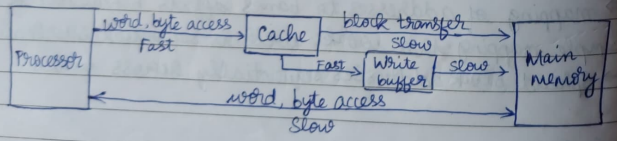
7/2/24

(6) REDUCE MISS PENALTY : Early Restart & Critical Word First

- Don't wait for full block before restarting CPU
- CRITICAL WORD FIRST - Request missed word from memory first, send it to CPU right away; let CPU continue while filling rest of the block
- EARLY RESTART - As soon as requested word of block arrives, send to CPU and continue execution
  (Sequential access)

(7) MERGING WRITE BUFFER TO REDUCE MISS PENALTY

- Write buffer lets processor continue while waiting for write to complete
- Merging write buffer
  ⇒ If buffer contains modified blocks, addresses can be checked to see if new data matches that of some write buffer entry
  ⇒ If so, new data combined with that entry



(8) REDUCING MISSES BY COMPILER OPTIMIZATIONS SOFTWARE ONLY APPROACH

Instructions

→ REORDERING PROCEDURES in memory to reduce conflict misses

Data
- LOOP INTERCHARGE — Change nesting of loops to access data in memory order
- BLOCKING — Improve temporal locality by accessing blocks of data repeatedly

* Conflicts arise when both row and column majors are used simultaneously
{special PC = PC + x has to be defined, more cycles consumed}

8/2/24

HARDWARE PREFETCHING OF INSTRUCTIONS & DATA

- Hardware prefetches item before the processor requests them

STREAM BUFFER is used to store the next likely address to be requested

Compiler Controlled Prefetching {to reduce miss penalty/rate}

Compiler to insert prefetch instructions to request data before the processor needs it

Types of prefetch
- (i) Data prefetch
- (ii) Cache prefetch (load into cache)
- (iii) Speculative prefetch

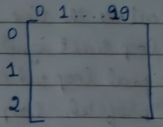
Question For the code below, determine which accesses are likely to cause data cache misses. Next, insert prefetch instructions to reduce misses. Finally, calculate the number of prefetch instructions executed and the misses avoided by prefetching.
Assume an 8KB direct mapped cache with 16 byte blocks is present and it is write back cache that does not write allocate. The elements of a and b are 8 bytes long since they are double-precision floating-point arrays.

a has 3 rows and 100 columns
b has 101 rows and 3 columns
Assume a and b are not in the cache at the start of the program.

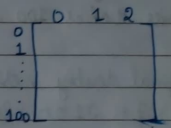
```
for (i=0; i<3;i=i+1)
    for(j=0; j<100; j=j+1)
        a[i][j] = b[j][0] * b[j+1]b[0];
```


Answer

a ranges from rows 0-2
columns 0-99



b ranges from rows 0-100
columns 0-2



Number of elements brought in during one access = $\frac{16}{8}$ = 2 elements

Total access = 3 × 100 = 300
Since only 2 elements are brought ⇒ even numbers lead to miss; odd numbers lead to hit

∴ Number of misses = $\frac{300}{2}$ = 150 misses

12/2/24

Compiler-Controlled Prefetching Example

```
for( j=0 ; j< 100 ; j=j+1) {
    prefetch ( b[j+7][0] );
    /* b(j, 0) for 7 iterations later */
    prefetch (a[0][j+7]);
    /* a(0,j) for 7 iterations later */
    a[0][j] = b[j][0] * b[j+1][0]; }
for (i=1 ; i<3; i=i+1)
    for (j=0 ; j < 100 ; j=j+1){
        prefetch(a[i][j+7]);
        /* a(i,j) for +7 iterations */
        a[i][j] = b[j][0] * b[j+1][0]; }
```

7 initially compulsory miss
4 miss i=0
4 miss i=1
4 miss i=2
19 misses

Example 2 (TB page 94-95) Calculate the time saved in Example 1.

Key loop times ignoring cache misses
- Original loop takes 7 clock cycles per iteration
- 1st prefetch loop takes 9 clock cycles per iteration
- 2nd prefetch loop takes 8 clock cycles per iteration

A miss takes 100 clock cycles

Original doubly nested loop → 3 × 100 = 300 times execution (i<3, j<100)

300 × 7 = 2100 clock cycles + cache misses
(7 cc per iteration)

Cache misses → 251 × 100 = 25100 clock cycles
(101+150 for b, for a; 100 cc per miss)

Total = 25100 + 2100 = 27200 clock cycles

1st prefetch loop ⇒ iterates 100 times ⇒ 900 clock cycles + cache misses
(7+4 = 11 × 100 = 1100 cc)

2nd prefetch loop ⇒ i = 1, 2 ; j = 0, ..., 99 (2 × 100 = 200 iterations) ⇒ 8 × 200 = 1600 clock cycles + cache misses
(4+4 = 8 × 100 = 800 cc)
= 2000 cc

= 2400 clock cycles

Total = 2000 + 2400 = 4400 clock cycles

If prefetch is overlapped with rest of execution

⇒ 27200 / 4400 = 68 / 11 = 6.1818 times faster

Virtual Memory Monitor (VMM)
→ Controlled access
→ Protection bits in resource sharing

user 1 { Process 1, Process 2 } Virtual memory (does not exist on physical memory)
Process n

Partitioning
~~Partioning~~
Paging
Segmentation

PAGING
- Consider a page size of 1KB
- A process of size 4.5 KB is partitioned into 5 pages
- 0.5 KB is unused in the last page → cannot be used by another process since processes are independent of each other

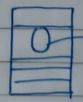
* Unused memory within a page ⇒ INTERNAL FRAGMENTATION

A page size of 4KB would result in an internal fragmentation of 3.5KB for a process of 4KB size

★ Internal fragmentation can be reduced by taking smaller page sizes

SEGMENTATION

~~Hole~~ HOLE/EXTERNAL FRAGMENTATION
↳ reduced by method of COMPACTION
↳ overhead



PAGING

- Each process is divided into small fixed-size chunks of the same size known as PAGES {associated with virtual memory}
- Pages assigned to available chunks of memory known as FRAMES or PAGE FRAMES {associated with main memory}

★ Page size = Frame size

★ PAGE TABLE contains mapping details about page number and frame number

page# | Frame#

- initially made to reside in main memory
- each active process has a page table loaded onto main memory
  → still occupies some amount of space in main memory
  ⇒ Use Inverted Page Table instead
  ↳ maintained in Virtual memory

Logical address

Page #	Offset

Offset of 10 bits ⇒ 2^{10} combinations
↳ 1024 ⇒ 1KB page size ⇒ can address from 0 to 1023

Example Assume 16-bit addressing and the page size as 1KB. Given the relative address as 2024 find the page number and offset (i.e. the logical address)

Page Size 1KB → 2^{10} ⇒ 10 bits for offset
2024 - 1024 = 1000 → offset value
2024 ⇒ 0X7E8

* nothing specified → ~~convert~~ address given in decimal (default)

Logical address 0000 01|1 1110 1000
page # | offset

page # = 1

If

page#	frame#
0	0
1	7
.	.

Page # 1 is mapped to frame # 7
⇒ 7KB with offset 1000 in main memory

0111 01|1 1110 1000

Physical address if frame no. 7 → (frame no. × page size) + offset
7KB + 1000
= 0 X 1FE8

13/2/24

VIRTUAL ADDRESS TO PHYSICAL ADDRESS

Virtual page number	Page offset

* Page table is indexed with the virtual page number to obtain the corresponding portion of physical address
* Page table maps each page in virtual memory to either a page in main memory or a page stored on disk (next level in hierarchy).

ASIDE
TRANSLATION LOOK-AHEAD BUFFER (TLB)
→ acts as a cache of the page table for the entries that map to physical pages only
→ overall access time is reduced

14/2/24

Q1. Consider the single-level paging scheme. The virtual address space is 4MB and page size is 4KB. What is the maximum page table entries size possible such that the entire page table fits well in one page?

Virtual memory space = 4MB
Page size = 4KB
$\frac{4 \times 2^{20}}{4 \times 2^{10}} = 2^{10} = 1024$ entries

* For page table to fit well in one page
page table size $\leq$ page size

Let page table entry size = B bytes

Page table size = Number of entries in page table × Page table entry size

$2^{10} \times B$ bytes ≤ 4KB

$B \leq \frac{4 \times 2^{10}}{2^{10}}$

$B \leq 4$ max possible page table entry size = 4 bytes

Q2. Consider a machine with 64 MB physical memory and a 32 bit virtual address space. If the page size is 4 KB, what is the approximate size of the page table?

Physical memory = 64MB
Page size = 4KB
Number of frame = $\frac{64\text{MB}}{4\text{KB}} = \frac{2^6 \times 2^{20}}{2^2 \times 2^{10}} = 2^4 \times 2^{10} = 16384$ frames
⇒ 14 bits required to represent frame number
Size of virtual memory = 2^{32} → 32-bit addressing
Number of entries = $\frac{2^{32}}{2^2 \times 2^{10}} = 2^{20}$ entries

B bytes per page
$2^{20} \times B$ bytes $\leq$ 4KB

Page table size = Number of entries in page table × Page table entry size = No. of page table entries × No. of bits in frame number
$= 2^{20} \times 14$ bits
$\simeq 2^{20} \times 16$ bits (byte-oriented)
$= 2^{20} \times 2$ bytes
$= 2$ MB

Q3. Compare the access times of 64KB caches with 64B blocks and a single bank. What are the relative access times of 2-way and 4-way set associative caches in comparison to a direct-mapped organisation? Assume the access time of
- direct-mapped cache is 0.86 ns
- 2-way cache is 1.12 ns
- 4-way cache is 1.37 ns

$\frac{\text{2-way}}{\text{direct-mapped}} = \frac{1.12\text{ns}}{0.86\text{ns}} = 1.302$

$\frac{\text{4-way}}{\text{direct-mapped}} = \frac{1.37\text{ns}}{0.86\text{ns}} = 1.593$

* Access time for direct-mapped is the least
* Access time usually increases as associativity is increased

Q4. Compare the access times of 4-way set associative caches with 64 bytes and a single bank. What are the relative access times of 32KB and 64KB caches in comparison to a 16KB cache.

Given Access time of 16KB → 1.27ns
32KB → 1.35ns
64KB → 1.37ns

$\frac{\text{32KB cache}}{\text{16KB cache}} = \frac{1.35\text{ns}}{1.27\text{ns}} = 1.062$

$\frac{\text{64KB cache}}{\text{16KB cache}} = \frac{1.37\text{ns}}{1.27\text{ns}} = 1.0787$

Q5. For a 64 KB cache, find the cache associativity between 1 and 8 with the lowest average memory access time, given that misses per instruction for a certain workload suite is DM → 0.00664
2 → 0.00366
4 → 0.000987
8 → 0.000266

Overall, there are 0.3 references per instruction. Assume cache misses take 10 ns for all models.
Cycle time – 0.5 ns (DM, 2-way)
– 0.83 ns (4-way)
– 0.79 ns (8-way)

$$\text{Miss rate} = \frac{\text{Misses per instruction}}{\text{References per instruction}} \times 100$$

DM → $\frac{0.00664}{0.3} \times 100 = 2.213\%$. 4-way → $\frac{0.000987}{0.3} \times 100 = 0.329\%$

2-way → $\frac{0.00366}{0.3} \times 100 = 1.22\%$. 8-way → $\frac{0.000266}{0.3} \times 100 = 0.08867\%$

$t_{DM} = 0.5 + \frac{2.213}{100} \times 10 = 0.7213$ ns

$t_{2\text{-way}} = 0.5 + \frac{1.22}{100} \times 10 = 0.622$ ns

$t_{4\text{-way}} = 0.83 + \frac{0.329}{100} \times 10 = 0.8629$ ns

$t_{8\text{-way}} = 0.79 + \frac{0.08867}{100} \times 10 = 0.798867$ ns

∴ 2-way cache associativity has the least/lowest average memory access time

15/2/24

ALIASING – A situation in which 2 addresses access the same object
– Can occur in virtual memory when there are 2 virtual addresses for the same physical page

SEGMENTATION

- Dividing the program and its associated data into partitions of equal or unequal length
- A logical address using segmentation consists of 2 parts, a SEGMENT NUMBER and an OFFSET.
- Similar to dynamic partitioning
  Different in terms of → a pgm may occupy more than one partition, and these partitions need not be contiguous

★ offset < Length of Process Segment Table for the particular segment
↳ physical address obtained

★ Offset > Length ⇒ Segmentation fault

Q. Assume 4KB pages, a 4-entry fully associative TLB, and true LRU replacement. If the following references are made 4669, 2227, 13916, 34587, 48870, 12608, 49225
Identify the TLB Miss, page fault and the final page table entry at the end of all the accesses
Assume that the if the pages must be brought in from disk, increment the next largest page number.

TLB (Same as page table (in fully associative (∵ only tag & offset fields))

VALID	TAG	PHYSICAL PAGE NUMBER
1	~~4~~ 0	~~12~~ 5
1	~~7~~ 8	~~1~~ 14
1	3	6
~~0~~ 1	~~X~~ 11	~~8~~ 13 12

TLB Miss → no page fault
→ there is mapping in page table

Page Table

	VALID	PHYSICAL PAGE / IN DISK
0	1	5
1	~~0~~ 1	~~Disk~~ 13
2	0	Disk
3	1	6
4	1	9
5	1	11
6	0	Disk
7	1	4
8	~~0~~ 1	~~Disk~~ 14
9	0	Disk
10	1	3
11	1	12

Page size = 4KB
$= 2^{12}$
⇒ 12 bits for offset

4669 ⇒ 0x123D
Page number = 1
TLB Miss
Disk; V=1 nhi page = 13

2227 ⇒ 0x08B3
Page number = 0

13916 ⇒ 0x365C
Page number = 3

20/2/24

* 4669 ⇒ 0x123D Page number = 1 ⇒ TLB Miss
Go to Page Table → at index 1, V=0 ⇒ page present in disk
Replace Disk by 13 and set V=1 for index 1 as per the assumption given in the question
One of the entries of TLB has V=0
Set that V=1, Tag=1, Physical Page number = 13

Physical memory address generation 13 ⇒ 0xD
∴ Physical memory address → 0xD23D

* 2227 ⇒ 0x8B3 Page number = 0 ⇒ TLB Miss
Page Table index 0 → V=1 ⇒ page present at 5
Using LRU, replace Tag=11 with Tag=0 and physical page number to 5
Physical memory address ⇒ 0x58B3

* 13916 ⇒ 0x365C Page number = 3 ⇒ TLB Hit
Physical memory address → 0x665C

* 34587 ⇒ 0x871B Page number = 8 ⇒ TLB Miss
Go to Page Table → at index 8, V=0 → Set V=1 and physical page to 14 [brought in] ↳ Page fault
In TLB, replace second entry to Tag=8, Physical page no.=14
Physical memory address → 0xE71B

* 48870 ⇒ 0xBEE6 Page number = ~~12~~ 11 ⇒ TLB Miss
Page Table → index 11, V=1 ⇒ No page fault
Replace Tag=1 to Tag=11 Physical mem addr ⇒ 0xCEE6

* 12608 ⇒ 0x3140 Page number = 3 ⇒ TLB Hit
Physical memory address → 0x~~5~~3140

* 49225 ⇒ 0xC049 Page number = 12 ⇒ TLB Miss
Page Table → no entry 12 ⇒ Disk miss
Add entry at index 12 to Page Table V=1, Physical page = 15

Decode ⇒ identifies the operation and operands involved

Pipeline CPI = Ideal pipeline CPI (cycles per instruction) + Structural Stalls + Data hazard Stalls + Control Stalls

	1	2	3	4 (buffer)	5	6	7	8	9	10	11
Sub S2, S1, S3	IF	ID	EX	MEM	WB → not written back to S2 by sub						
and S12, S2, S5		IF	ID	Stall	Stall	EX	MEM	WB → S2 in use			
or S13, S6, S2			IF	ID	Stall	Stall	EX	MEM	WB		
add S14, S2, S2				IF	ID	Stall	Stall	EX	MEM	WB	
sw S15, 100(S2)					IF	ID	Stall	Stall	EX	MEM	WB

Rearrangement
Removing redundancy

DEPENDECES → DATA DEPENDENCES (TRUE DATA DEPENDENCES)
DEPENDECES → NAME DEPENDENCES
DEPENDECES → CONTROL DEPENDENCES

Instruction j dependent on instruction i if
- i produces a result that may be used by j
- j is data dependent on instruction k & instruction k is data dependent on instruction i

Data dependence conveys 3 things
- possibility of a hazard
- order in which results must be calculated
- an upper bound on how much parallelism can possibly be exploited

Dependence overcome by
- maintaining the dependence but avoiding a hazard
- eliminating a dependence by transforming the code (rename registers being used)

21/2/24

NAME DEPENDENCE

- occurs when 2 instructions use the same register or memory location (NAME), but there is no flow of data between the instructions associated with that name

Types of name dependences b/w instr. i that precedes instr. j
- antidependence ⇒ j writes a reg or memory location that i reads (no true dependence → no flow of data) ⇒ parallel or reordering not possible
- output dependence ⇒ i and j write the same reg or mem location

- does not exist due to data transfer b/w instructions; exists because of the memory locations being accessed.

* Can be resolved by RENAMING (done)
  - statically by compiler
  - dynamically by hardware

DATA HAZARD exists whenever there is a name or data dependence between instructions

Possible data hazards:

(i) RAW ⇒ Read After Write
- i: R2 ← R1 + R3
- j: R4 ← R2 + R3

- true data dependence
- program order must be preserved to ensure j receives correct value of R2 after execution of i

(ii) WAW ⇒ Write After Write
- i: R2 ← R1 + R3
- j: R2 ← R4 + R5

(iii) WAR (Write After Read)

```
i   R2 ← R1 + R3
j   R3 ← R4 + R5
```

- arises from an antidependence

* There is no data hazard called RAR

CONTROL DEPENDENCE

Determines the ordering of an instruction i w.r.t a branch instruction (so that instr. i is executed in correct pgm order and only when it should be.

```
if p1{          S1 is control dependent on p1
  S1;           S2 is control dependent on p2 but not on p1
};
if p2{
  S2;
}
```

Constraints imposed by control dependences

- an instr. that is control dependent on a branch cannot be moved before the branch so that its execution is no longer controlled by the branch
- an instruction that is not control dependent on a branch cannot be moved after the branch so that its execution is controlled by the branch

When processors preserve strict pgm order, they ensure that control dependences are also preserved

Speculation
- SOFTWARE SPECULATION
- HARDWARE SPECULATION

Control dependence is preserved by implementing control hazard detection that causes CONTROL STALLS.
Controll stalls can be eliminated or reduced by a variety of hardware and software techniques

LOOP UNROLLING

Latencies of FP Operations Used

Instr. producing result	Instr. using result	Latency (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

```
for (i = 999 ; i >=0 ; i = i-1)
    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 ptr by 8 bytes
       BNE     R1,R2, Loop   ; branch R1 != R2
```

23/2/24

Since it is floating point type data, each occupies 8 bytes
(The assembly code to write before scheduling)

Delay → 1+2+0+1

Without scheduling

Loop :
LD
Stall
ADD.D
Stall
Stall
S.D
DADDUI
Stall
BNE

RAW
RAW
dependency exists

∴ Loop unrolling is done

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

7 cycles with scheduling (only 2 stalls)

* Stalls after ADD.D are used by S

```
Loop: LD   F0, 0(R1)
      LD   F6, -8(R1)
      LD   F10, -16(R1)
      LD   F14, -24(R1)

      ADD.D F4, F0, F2
      ADD.D F8, F6, F2
      ADD.D F12, F10, F2
      ADD.D F16, F14, F2

      SD   F4, 0(R1)
      SD   F8, -8(R1)
      DADDUI R1, R1, #-32
      SD   F12, 16(R1)
      SD   F16, 8(R1)
      BNE  R1, R2, LOOP
```

21 cycles for 4 set of instructions