Introduction to the

MIPS R10000

Processor

Mark Portney

mnp@sgi.com

microprocessor pipelines

MIPS R10000 microprocessor

"porting" to the R10000?

comparison to other MIPS micros using high energy physics codes

Microprocessor Pipelines

Pipelines

Integer Pipeline / Integer Memory Pipeline

F	D	A	E	W
instruction fetch alignment predecode branch prediction	decode scoreboard register file read	generate load/store addresses	ALU data cache TLB lookup branch resolution exception detection	reg file write

latency if long alu ops

Floating Point Memory Pipeline

F	D	A	E	...	W
instruction fetch alignment predecode branch prediction	decode scoreboard register file read	generate load/store addresses	TLB lookup	5 cycle latency	write to LDQ in FPU

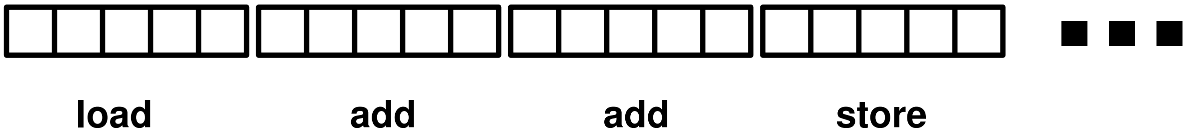
Floating Point Pipeline

F	D			
instruction fetch alignment predecode branch prediction	decode scoreboard register file read	stage to fpu	stage to fpu	write to floating point queue (FPQ)

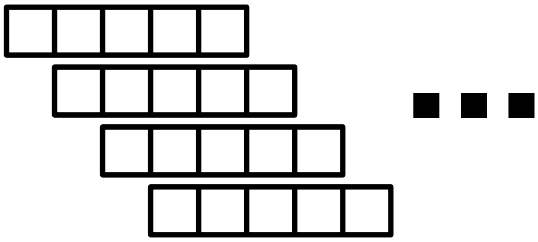
decoupled

sometime later

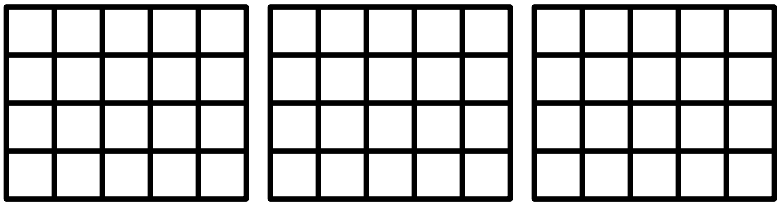
...						W
	floating point issue	read from LDQ	register file read register file write	fp short ops (mov,abs,...) fp execute stage 1	... latency	write to reg file



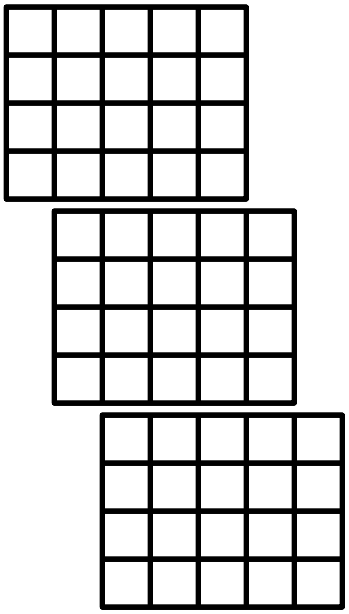
not pipelined



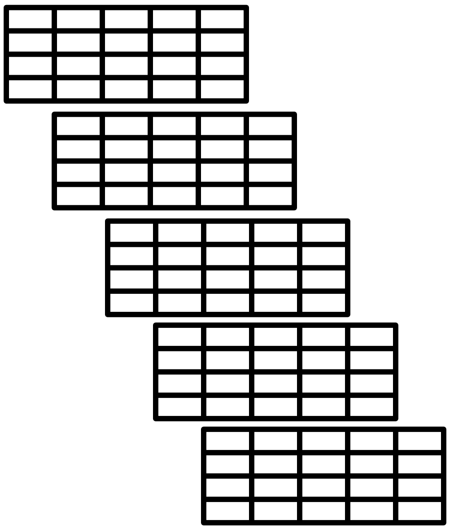
pipelined



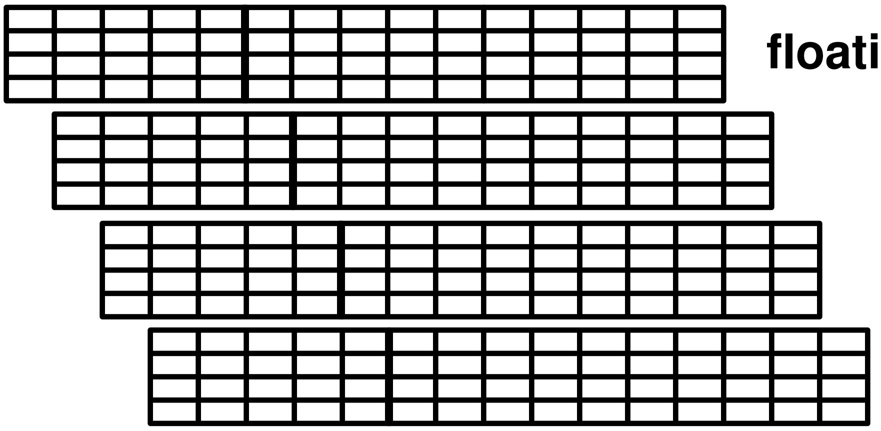
superscalar
not pipelined



superscalar
and pipelined



not as simple as this looks, due to different lenth pipelines (int, mems, flt) and decoupled FPU



floating point looks more like this

Optimization

optimal packing of superscalar pipeline to minimize total length

compiler is responsible for most of this

user must help compiler

compiler uses software pipelining to hide latencies

Software Pipelining

Loops are where most of the time is spent

Loops have parallelism

```
do i = 1, n
   y(i) = y(i) + a*x(i)
enddo
```

2 loads, 1 store
1 multiply−add
2 address increments
1 loop−end test
1 branch

Compiler needs to fill up those 4 superscalar slots to maximize performance

Long (5 cycle) latency to Gcache

Integer Unit is many (6) cycles ahead of Floating Point Unit

Software Pipelining

fill up superscalar instruction slots

AND

hide latency between integer unit and Gcache and floating point unit

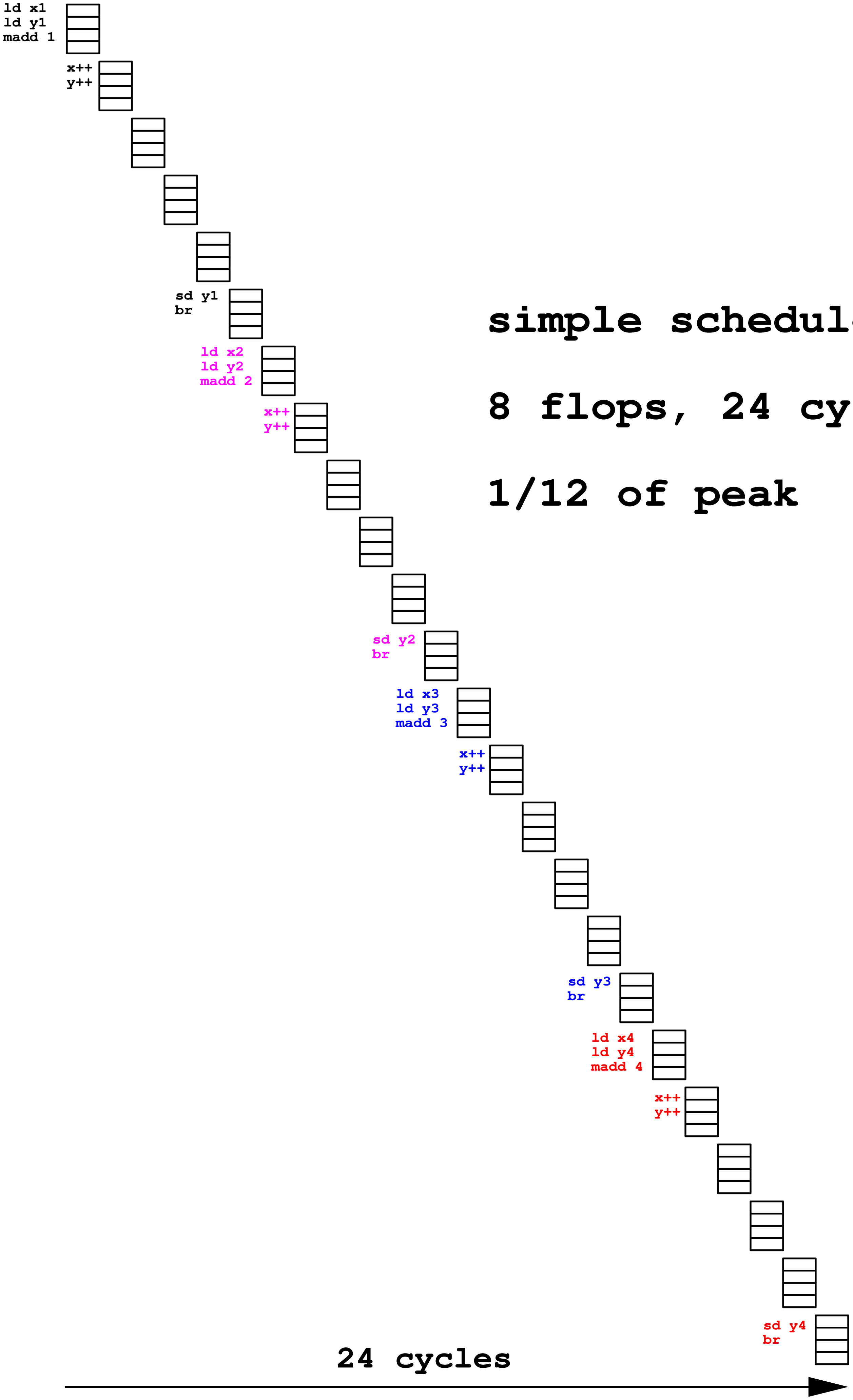
SO THAT

total pipeline length is shortest

software pipelining

example: daxpy

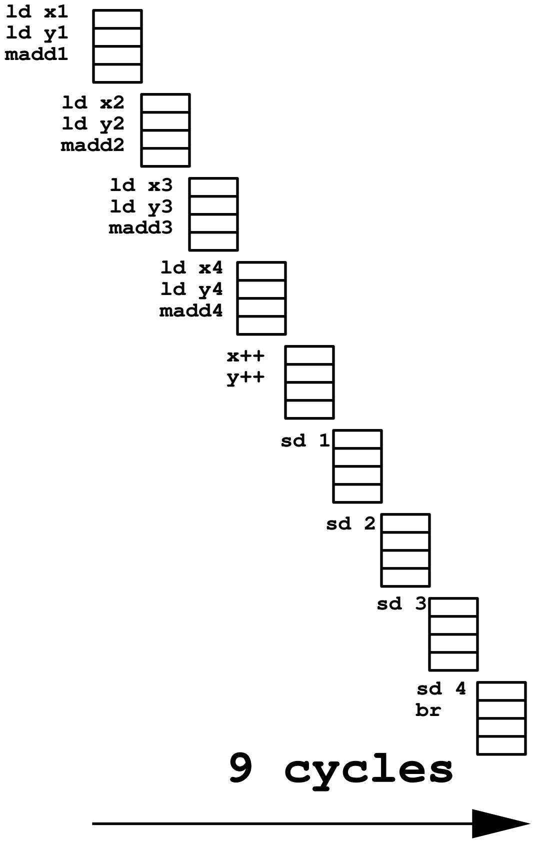
```
do i = 1,n
   y(i) = y(i) + a*x(i)
end do
```



simple schedule

8 flops, 24 cycles

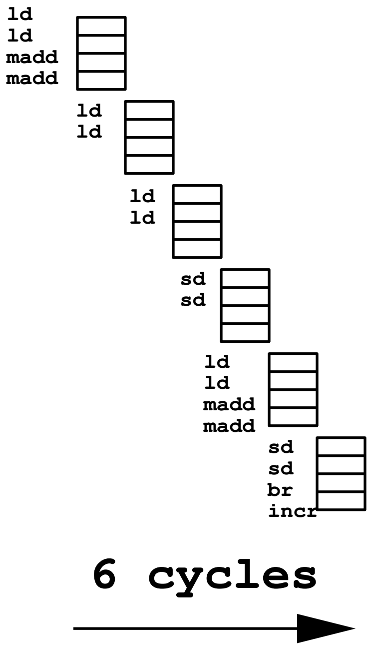
1/12 of peak



unrolled x 4

8 flops, 9 cycles

2/9 of peak

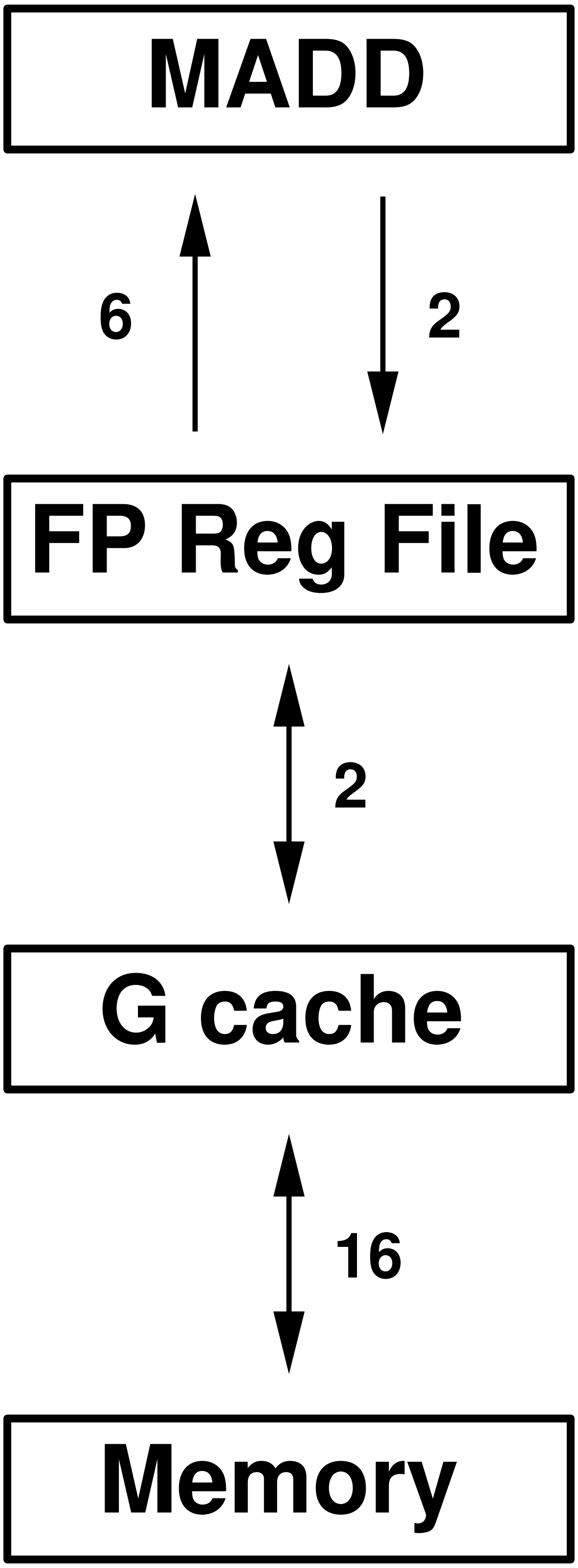


software pipelined

8 flops, 6 cycles

1/3 of peak

Memory Hierarchy for Floating Point



6 and 2 dw/clock

5.76 GB/sec

no latency

2 dw/clock

1.44 GB/sec sustained

5 cycle latency, pipelined

16 dw/miss

~ 170 MB/sec sustained

66 cycle latency

note: dw = doubleword (8 bytes)

Optimization for Gcache

don’t miss it!

66 cycle latency to memory

average 4 clocks per double precision miss if entire line used

average 2 clocks per single precision miss if entire line used

Optimization for Memory Hierarchy

reuse data in registers

perform as many operations as are possible on a piece of data once it is in the registers

reduce number of loads/stores to/from registers

reuse data in Gcache

perform as many operations as are possible on a piece of data once it is in the Gcache

MIPS R10000 Architecture

Porting to the R10000

ISA’s, 64 bits, ABI’s, Compilers

ISA – Instruction Set Architecture

defines instructions available to user process

defines memory organization visible to user process

defines register set

ISA is independent of the implementation

mips1 – base MIPS instruction set

single cycle issue

small number of fixed length instructions

load / store architecture

large number of registers

mips2

- eliminated load−delay slot requirement
- added co−processor (FPU) 64 bit ld/st
- ll/sc MP synchronization primitives
- sync instruction
- branch likely
- trap on condition
- square root

mips3

- 64 bit integers (64 bit integer registers, 64 bit ld/st for integers, 64 bit integer and logical operations, 64 bit integer−FPU moves
- added 16 64−bit FP registers

mips4

- conditional moves
- floating point multiply−add (madd)
- floating point indexed addressing
- reciprocal square root
- multiple floating point conditional codes
- imprecise exceptions
- prefetch

ABI – Application Binary Interface

defines how instruction set used in an application

- passing arguments in registers
- making gp register callee–saved
- removing exit block restrictions
- requiring callee to reserve register–save space

allow compilation system to make more aggressive optimizations

3 MIPS ABI's

–32

"old" ABI

mips2

32 bit addresses

usually very inefficient on modern cpus

–n32

mips4

32 bit addresses

–64

mips4

64 bit addresses

only needed for addressing > 2 GBytes

possible porting problems from 32 bit

porting from 32 bit to 64 bit

C – problems if either long or pointer was assumed to be 32 bit

Fortran – problems only if Fortran extension of pointers is used, and passed as integers

Fortran calling C – often used for memory management; passing pointers from C to Fortran that are then stored in integers; big headache compiler's role in ISA, ABI

choose ABI

choose ISA

set default size on integers and reals for Fortran

use / don't use certain instructions

reciprocal, reciprocal square root

IEEE arithmetic ?

roundoff ?

32 bit vs. 64 bit

what does 64 bit mean?

- **data types**
  - **Fortran – integer, real**
  - **C – int, long**
- **addresses**
  - **Fortran extensions**
  - **C pointers**
- **arithmetic**

Fortran: addresses are 64 bit

C: LP64 (longs, pointers are 64 bit)

64 bit pointers are only needed to address beyond 2 GB

64 bit pointers almost same performance as 32 bit pointers

64 bit integer multiply, divide much slower than 32 bit

64 bit floating add, multiply usually same as 32 bit

64 bit divides, square roots much slower than 32 bit

64 bit makes caches look half as large as 32 bit

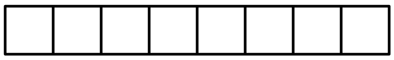
64 bit makes cache miss penalties longer than 32 bit

===> contrary to popular belief, 32 bit arithmetic is not slower than 64 bit arithmetic

Comparisons to other MIPS Microprocessors

MIPS R4400

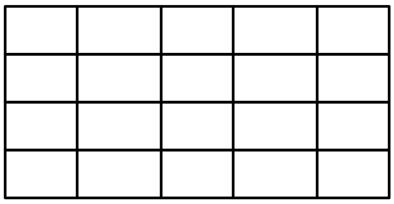
250 MHz



superpipelined,
not superscalar
no out-of-order
1/2 flops/clock
mips2

MIPS R8000

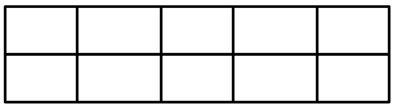
90 MHz



4-way superscalar
limited out-of-order
IU runs ahead of FPU
multi-chip set
4 flops/clock
mips4

MIPS R5000

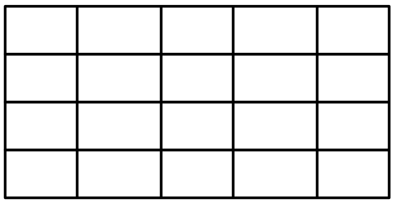
180 MHz



2-way superscalar if
one is floating point
no out-of-order
1 flop/clock double
2 flops/clock single
mips4

MIPS R10000

195 MHz



4-way superscalar
out-of-order
non-blocking caches
prefetch
2 flops/clock
mips4

vector add $z(i) = x(i) + y(i)$

(unrolled 8 times, 24 mem refs, 8 adds)

cpu	cycles	cyc/add	%peak	MHz	MFlops
r4k	33	4.125	48	250	60.6 *
r8k	12	1.5	16	90	60.0
r5k	25	3.125	15	180	57.6
r10k	24	3.0	16	200	66.7

* hard for compiler to get this for r4k

ktev

			−O0 ? mips2	−O2 mips2	−O3 mips4
250	MHz	R4400	5.97	3.63	
90	MHz	R8000	9.02	5.42	4.65
180	MHz	R5000	7.34	4.64	5.32
195	MHz	R10000	3.88	2.45	2.45

time in sec for 10000 events

tiny (recon)

			mips2	mips4
250	MHz	R4400	124	
90	MHz	R8000	70	88
180	MHz	R5000	120	132
195	MHz	R10000	192	252

VUPS (bigger is better)

mcnp

			mips2	mips4
250	MHz	R4400	38.9	
90	MHz	R8000	58.1	40.2
180	MHz	R5000	40.8	31.0
195	MHz	R10000	19.2	13.2

time in sec

4 pt convolution

			mips2	mips4
250	MHz	R4400	6.28	
90	MHz	R8000	10.68	2.36
180	MHz	R5000	6.02	2.43
195	MHz	R10000	3.16	2.12

time (sec)