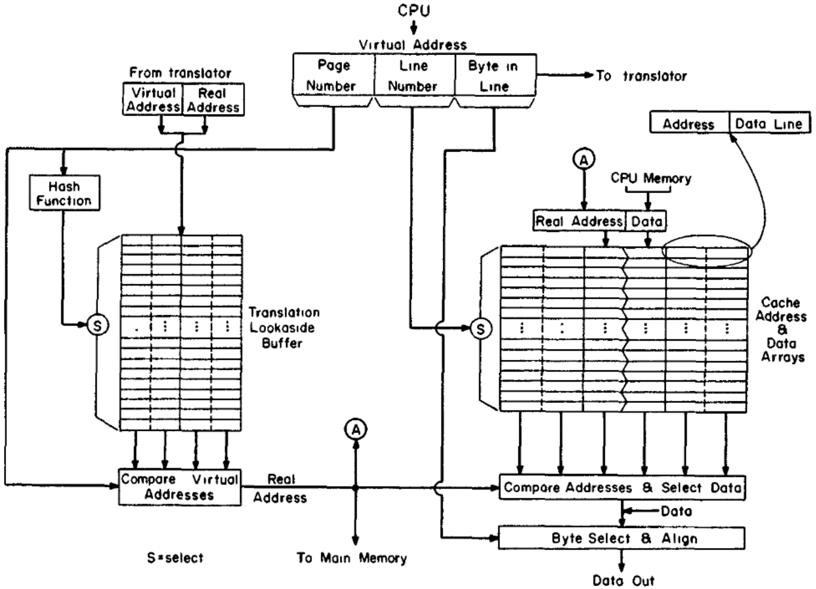
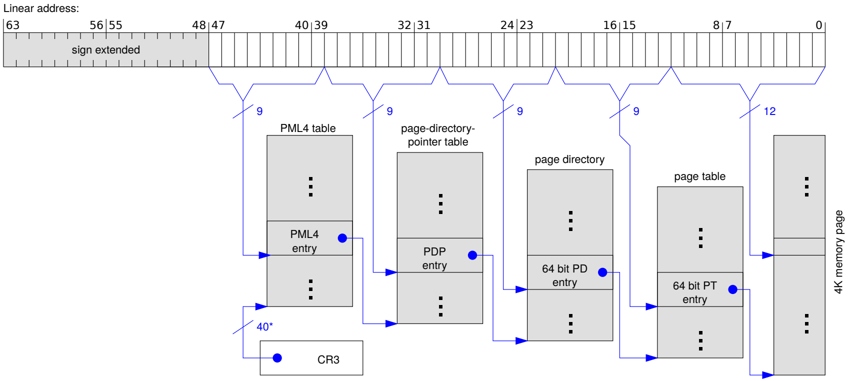
CS 6354: Pipelining / ISAs

7 September 2016

Review: Memory Hierarchy



Review: Page Tables



*) 40 bits aligned to a 4-KByte boundary

Review: Memory Hierarchy Optimizations

adjust # caches, sizes, associativity, block size, …

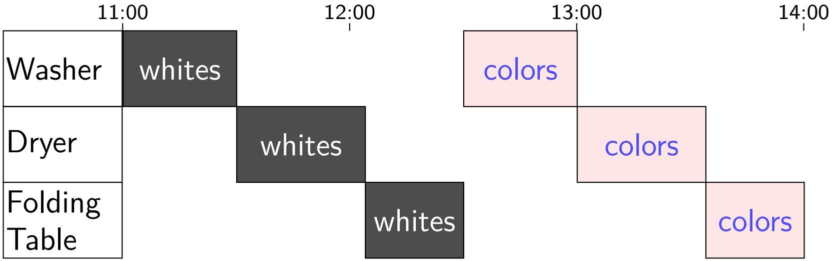
adjust when virtual to physical translation happens

add victim caches, prefetching, etc.

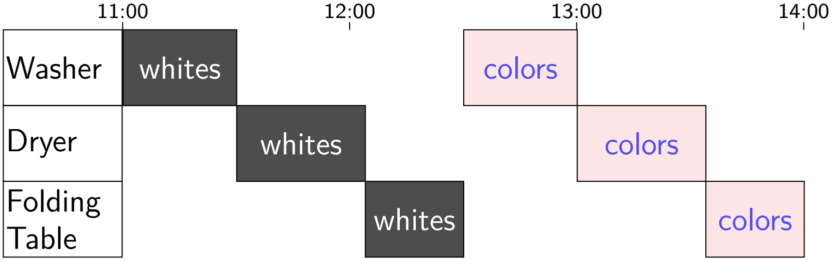
cache blocking — reorder code for more reuse

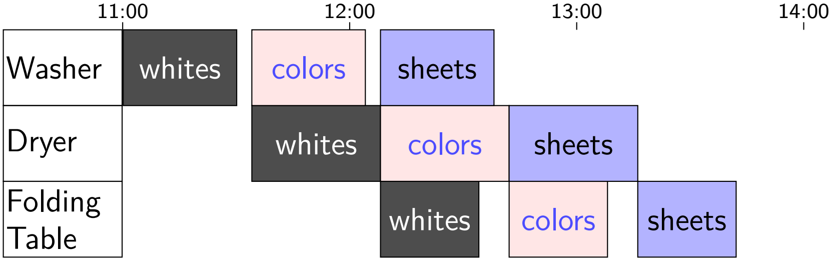
overlap memory accesses and

Human pipeline: laundry



Human pipeline: laundry





The MIPS pipeline

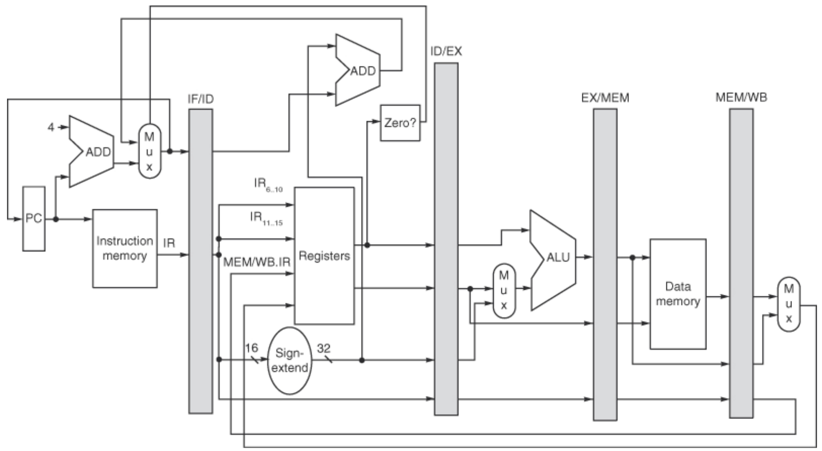


Figure C.28 The stall from branch hazards can be reduced by moving the zero test and branch-target calculation into the ID phase of the pipeline. Notice that we have made two important changes, each of which removes 1 cycle from the 3-cycle stall for branches. The first change is to move both the branch-target address calculation and the branch condition decision to the ID cycle. The second change is to write the PC of the instruction in the IF phase, using either the branch-target address computed during ID or

MIPS instruction execution (1)

```
add $1, $2, $3 ; reg[1] <— reg[2] + reg[3]
```

Instruction Fetch: read from instruction cache

Instruction Decode: read registers 2 and 3

Execute: compute reg[2] + reg[3]

Memory: do nothing

Write Back: write computed value into reg[1]

MIPS instruction execution (2)

```
sw r1, 100(r3) ; memory[100 + reg[3]] = reg[1]
```

Instruction Fetch: read from instruction cache

Instruction Decode: read registers 1 and 3

Execute: compute 100 + reg[3]

Memory: store reg[1] into data @ 100 + reg[3]

Write Back: do nothing

The MIPS pipeline

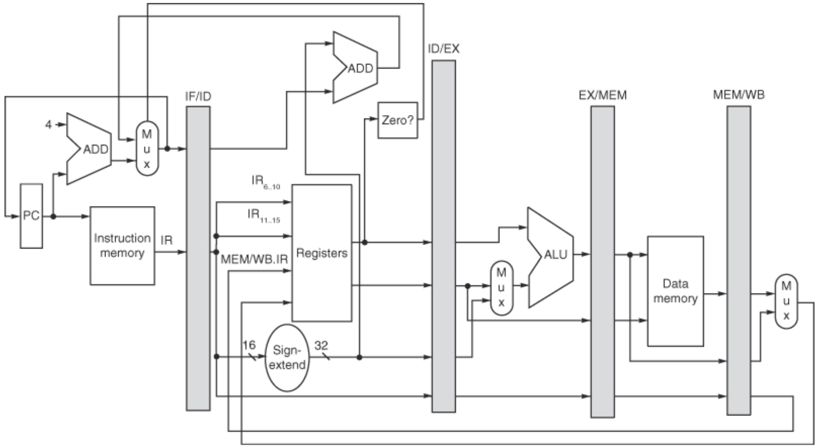


Figure C.28 The stall from branch hazards can be reduced by moving the zero test and branch-target calculation into the ID phase of the pipeline. Notice that we have made two important changes, each of which removes 1 cycle from the 3-cycle stall for branches. The first change is to move both the branch-target address calculation and the branch condition decision to the ID cycle. The second change is to write the PC of the instruction in the IF phase, using either the branch-target address computed during ID or

MIPS instruction execution (1)

```
add $1, $2, $3 ; reg[1] <- reg[2] + reg[3]
```

Instruction Fetch: read from instruction cache

    IF/ID stores: instr., PC

Instruction Decode: read registers 2 and 3

    ID/EX stores: reg[2], reg[3], instr., PC

Execute: compute reg[2] + reg[3]

    EX/MEM stores: reg[2] + reg[3], instr., PC

Memory: do nothing

    MEM/WB stores: reg[2] + reg[3], instr., PC

Write Back: write computed value into reg[1]

MIPS instruction execution (2)

```
sw r1, 100(r3) ; memory[100 + reg[3]] = reg[1]
```

Instruction Fetch: read from instruction cache

  IF/ID stores: instr., PC

Instruction Decode: read registers 1 and 3

  ID/EX stores: reg[1], reg[3], instr., PC

Execute: compute 100 + reg[3]

  EX/MEM stores: 100 + reg[3], reg[1], instr., PC

Memory: store reg[1] into data @ 100 + reg[3]

  MEM/WB stores: instr., PC

Write Back: do nothing

MIPS executing

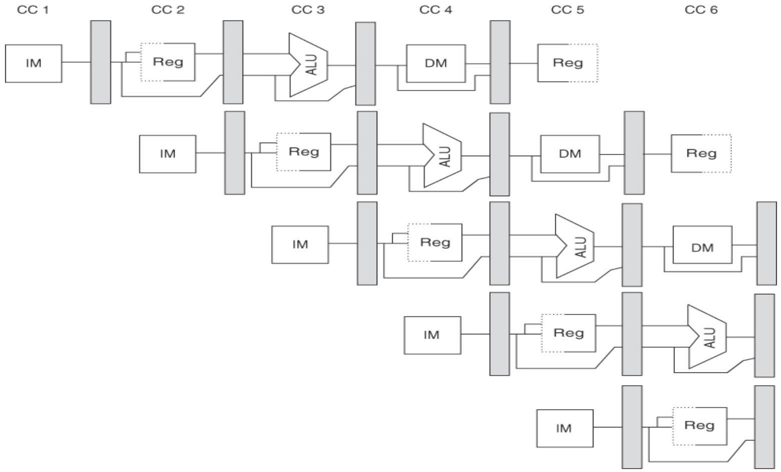


Figure C.3 A pipeline showing the pipeline registers between successive pipeline stages. Notice that the registers prevent

Pipeline Hazards

hazards stop pipeline from executing at full rate

structural hazards — not enough hardware

data hazards — value not computed soon enough

control hazards — instruction to execute not known soon enough

Functional Hazards

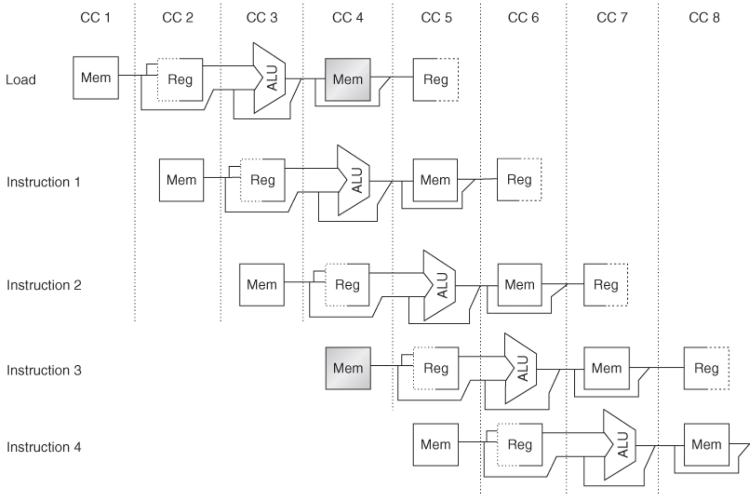


Figure C.4 A processor with only one memory port will generate a conflict whenever a memory reference occurs. In this

Figure: H&P Appendix C

Read-after-Write

```
add r1, r2, r3  ; r1 <— r2 + r3
sub r4, r1, r5  ; r5 <— r1 — r5
```

	`add r1, r2, r3`		`sub r4, r1, r5`	
1	*IF*			
2	*ID*:	read `r2`, `r3`	*IF*	
3	*EX*:	`temp1` $\leftarrow$ `r2` $+$ `r3`	*ID*:	read `r1`, `r5`
4	*MEM*		*EX*:	`temp2` $\leftarrow$ `r1 - r5`
5	*WB*:	`r1` $\leftarrow$ `temp`	*MEM*	
6			*WB*:	`r4` $\leftarrow$ `temp2`

Read-after-Write

```
add r1, r2, r3  ; r1 <— r2 + r3
sub r4, r1, r5  ; r5 <— r1 — r5
```

	`add r1, r2, r3`		`sub r4, r1, r5`	
1	*IF*			
2	*ID*:	read `r2`, `r3`	*IF*	
3	*EX*:	`temp1` $\leftarrow$ `r2` $+$ `r3`	*ID*:	read `r1`, `r5`
4	*MEM*		*EX*:	`temp2` $\leftarrow$ `r1 - r5`
5	*WB*:	`r1` $\leftarrow$ `temp`	*MEM*	
6			*WB*:	`r4` $\leftarrow$ `temp2`

Read-after-Write — Stall

```
add r1, r2, r3  ; r1 <- r2 + r3
sub r4, r1, r5  ; r5 <- r1 - r5
```

	add r1, r2, r3		sub r4, r1, r5	
1	*IF*			
2	*ID*:	read r2, r3	*IF*	
3	*EX*:	temp1 $\leftarrow$ r2 $+$ r3		stall
4	*MEM*			stall
5	*WB*:	r1 $\leftarrow$ temp1		stall
6			*ID*:	read r1, r5
7			*EX*:	temp2 $\leftarrow$ r1 $+$ r5
8			*MEM*	
9			*WB*:	r4 $\leftarrow$ temp2

Read-after-Write — Stall

```
add r1, r2, r3  ; r1 <- r2 + r3
sub r4, r1, r5  ; r5 <- r1 - r5
```

	`add r1, r2, r3`		`sub r4, r1, r5`	
1	*IF*			
2	*ID*:	read `r2`, `r3`	*IF*	
3	*EX*:	$\texttt{temp1} \leftarrow \texttt{r2} + \texttt{r3}$		stall
4	*MEM*			stall
5	*WB*:	$\texttt{r1} \leftarrow \texttt{temp1}$		stall
6			*ID*:	read `r1`, `r5`
7			*EX*:	$\texttt{temp2} \leftarrow \texttt{r1} + \texttt{r5}$
8			*MEM*	
9			*WB*:	$\texttt{r4} \leftarrow \texttt{temp2}$

Implementing Stalls

disable writing pipeline registers

need logic to detect conflicts

- function of pipeline registers (instruction values)

Read-After-Write

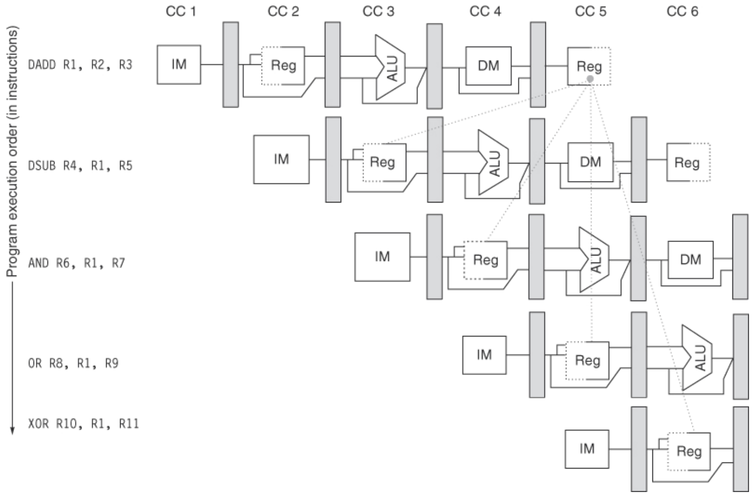


Figure C.6 The use of the result of the DADD instruction in the next three instructions causes a hazard, since the register is

Figure: H&P Appendix C

Read-after-Write — Forward

```
add r1, r2, r3  ; r1 <— r2 + r3
sub r4, r1, r5  ; r5 <— r1 — r5
```

	`add r1, r2, r3`		`sub r4, r1, r5`	
1	*IF*			
2	*ID*:	read `r2`, `r3`	*IF*	
3	*EX*:	`temp1` $\leftarrow$ `r2` $+$ `r3`	*ID*:	read `r1`, `r5`
4	*MEM*		*EX*:	`temp2` $\leftarrow$ `temp1` - `r5`
5	*WB*:	`r1` $\leftarrow$ `temp`	*MEM*	
6			*WB*:	`r4` $\leftarrow$ `temp2`

Forwarding

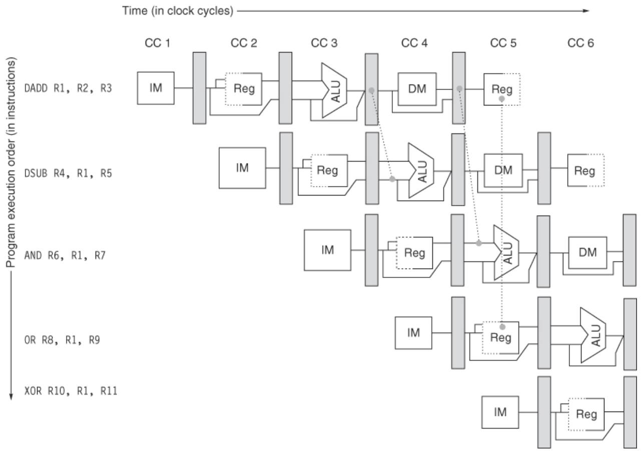


Figure C.7 A set of instructions that depends on the DADD result uses forwarding paths to avoid the data hazard. The inputs for the DSUB and AND instructions forward from the pipeline registers to the first ALU input. The OR receives its result by forwarding through the register file, which is easily accomplished by reading the registers in the second half of the cycle and

Figure: H&P Appendix C

Implementing Forwarding

multiplexers for operand values

need logic to detect which one to use

- function of pipeline registers (instruction values)

Implementing Forwarding

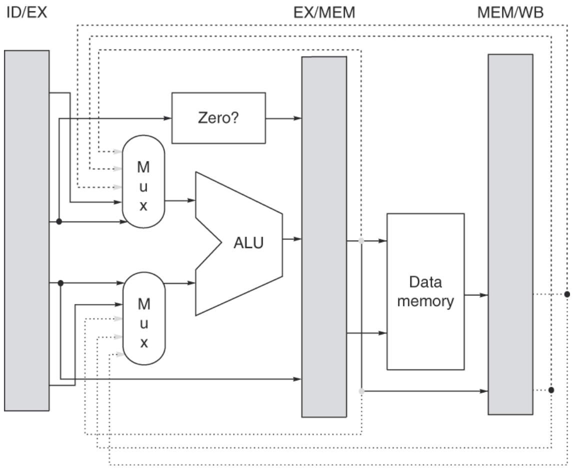


Figure C.27 Forwarding of results to the ALU requires the addition of three extra inputs on each ALU multiplexer and the addition of three paths to the new inputs. The paths correspond to a bypass of: (1) the ALU output at the end of the EX, (2) the

Limits of Forwarding

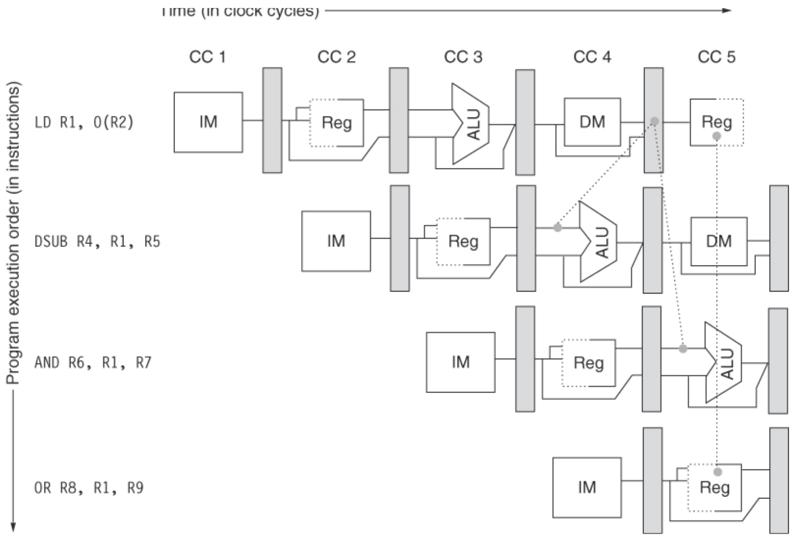


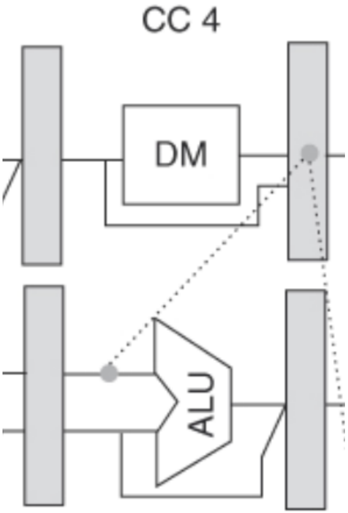
Figure: H&P Appendix C

Scheduling for Pipelines

```
lw  r1,  0(r20) ; r1 <— MEM[0+r20]
lw  r2,  4(r20) ; r2 <— MEM[4+r20]
add r3, r1, r2  ; r3 <— r1 + r2
lw  r4,  8(r20) ; r4 <— MEM[8+r20]
add r4, r4, r3  ; r4 <— r4 + r3
sw  r4,  8(r20) ; MEM[8+r20] <— r4
lw  r5, 12(r20) ; r5 <— MEM[12+r20]
mul r5, r5, r4  ; r5 <— r5 * r4
sw  r5, 12(r20) ; r5 <— MEM[12+r20]
```

Scheduling for Pipelines

```
lw   r1,  0(r20) ; r1 <— MEM[0+r20]
lw   r2,  4(r20) ; r2 <— MEM[4+r20]
add r3, r1, r2   ; r3 <— r1 + r2
lw   r4,  8(r20) ; r4 <— MEM[8+r20]
add r4, r4, r3   ; r4 <— r4 + r3
sw   r4,  8(r20) ; MEM[8+r20] <— r4
lw   r5, 12(r20) ; r5 <— MEM[12+r20]
mul r5, r5, r4   ; r5 <— r5 * r4
sw   r5, 12(r20) ; r5 <— MEM[12+r20]
```

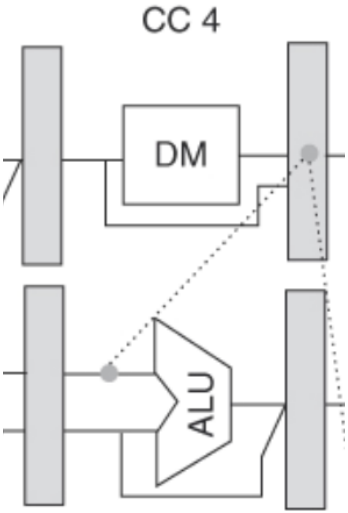


Scheduling for Pipelines

```
lw   r1,   0(r20) ; r1 <— MEM[0+r20]
lw   r2,   4(r20) ; r2 <— MEM[4+r20]
add  r3, r1, r2   ; r3 <— r1 + r2
lw   r4,   8(r20) ; r4 <— MEM[8+r20]
add  r4, r4, r3   ; r4 <— r4 + r3
sw   r4,   8(r20) ; MEM[8+r20] <— r4
lw   r5, 12(r20)  ; r5 <— MEM[12+r20]
mul  r5, r5, r4   ; r5 <— r5 * r4
sw   r5, 12(r20)  ; r5 <— MEM[12+r20]
```

converts into

```
lw   r1,   0(r20) ; r1 <— MEM[0+r20]
lw   r2,   4(r20) ; r2 <— MEM[4+r20]
lw   r4,   8(r20) ; r4 <— MEM[8+r20]
lw   r5, 12(r20)  ; r5 <— MEM[12+r20]
add  r3, r1, r2   ; r3 <— r1 + r2
add  r4, r4, r3   ; r4 <— r4 + r3
mul  r5, r5, r4   ; r5 <— r5 * r4
sw   r4,   8(r20) ; MEM[8+r20] <— r4
sw   r5, 12(r20)  ; r5 <— MEM[12+r20]
```

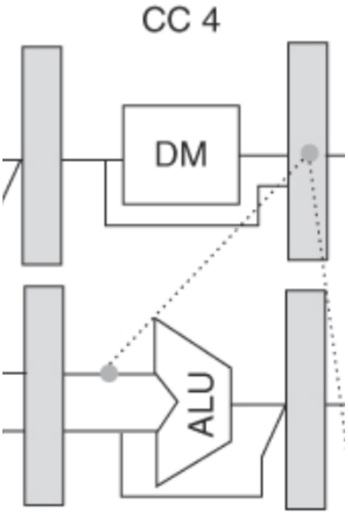


Scheduling for Pipelines

```
lw   r1,  0(r20) ; r1 <— MEM[0+r20]
lw   r2,  4(r20) ; r2 <— MEM[4+r20]
add r3, r1, r2   ; r3 <— r1 + r2
lw   r4,  8(r20) ; r4 <— MEM[8+r20]
add r4, r4, r3   ; r4 <— r4 + r3
sw   r4,  8(r20) ; MEM[8+r20] <— r4
lw   r5, 12(r20) ; r5 <— MEM[12+r20]
mul r5, r5, r4   ; r5 <— r5 * r4
sw   r5, 12(r20) ; r5 <— MEM[12+r20]
```

converts into

```
lw   r1,  0(r20) ; r1 <— MEM[0+r20]
lw   r2,  4(r20) ; r2 <— MEM[4+r20]
lw   r4,  8(r20) ; r4 <— MEM[8+r20]
lw   r5, 12(r20) ; r5 <— MEM[12+r20]
add r3, r1, r2   ; r3 <— r1 + r2
add r4, r4, r3   ; r4 <— r4 + r3
mul r5, r5, r4   ; r5 <— r5 * r4
sw   r4,  8(r20) ; MEM[8+r20] <— r4
sw   r5, 12(r20) ; r5 <— MEM[12+r20]
```



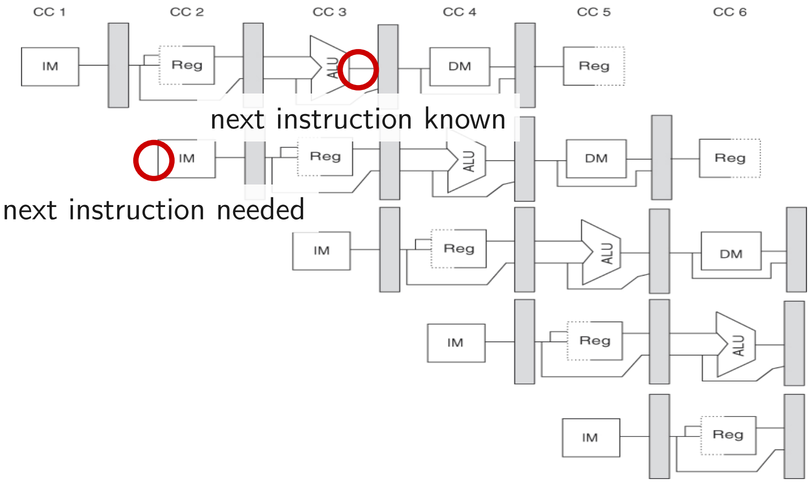
Next time: Scheduling

Weiss and Smith, “A study of scalar compilation techniques for pipelined supercomputers”

- theme: seperate dependencies from use
- focus on loops

Control Hazard

need to decode instruction to know next instruction



MIPS Delay Slots

avoid control hazard by delaying branch

```
    add $3, $4, $5          ; (1)
    beq $1, $2, label       ; (2)
    add $5, $6, $7          ; (3) DELAY SLOT
    add $6, $7, $8
    add $8, $9, $10
label:
    add $7, $8, $9          ; (4)
```

Branch Prediction

branch prediction — guess whether branch is taken

start guess immediately

clear pipeline registers if wrong

Speculation

when is it okay to guess

if we can undo guess if wrong

undo: clear pipeline registers before MEM, set new PC

Speculation

when is it okay to guess

if we can undo guess if wrong

MIPS pipeline:

- IF — doesn't change state
- ID — doesn't change state
- EX — doesn't change state
- MEM — changes memory!
- WB — changes registers!

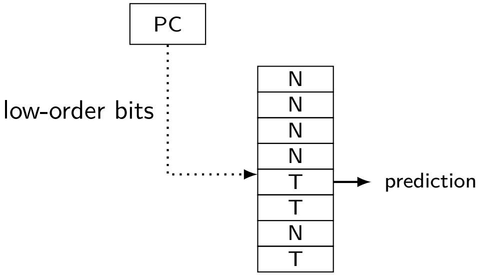
undo: clear pipeline registers before MEM, set new PC

Static branch prediction

forwards not taken (fetch normally)

backwards taken (fetch target)

Dynamic branch prediction

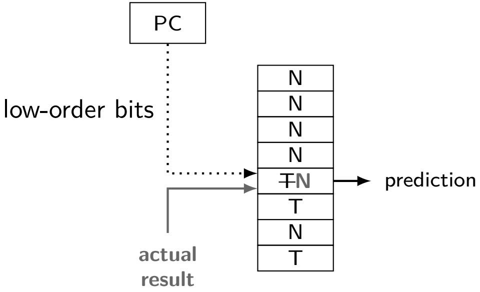


lookup branch address in table

1-bit: **T**aken/**N**ot taken

taken before ⇒ taken again

Dynamic branch prediction



lookup branch address in table

1-bit: **T**aken/**N**ot taken

taken before $\Rightarrow$ taken again

Dynamic branch prediction

refinement: 2 bits

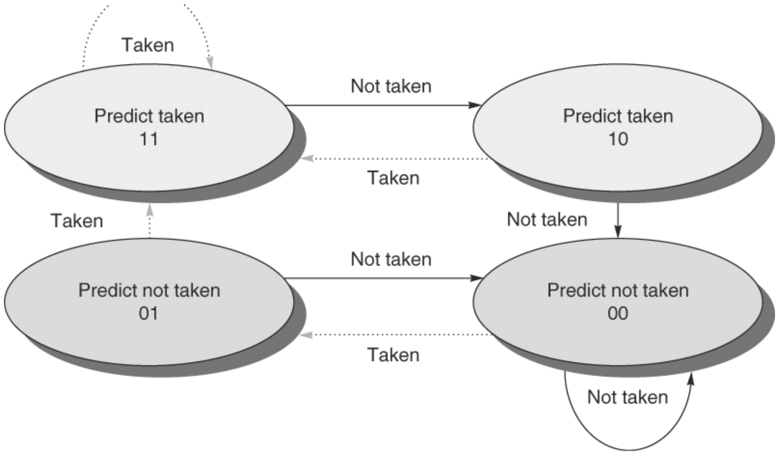


Figure: H&P Appendix C

Figure C.18 The states in a 2-bit prediction scheme. By using 2 bits rather than 1, a branch that strongly favors taken or not

Deeper Pipelines (1)

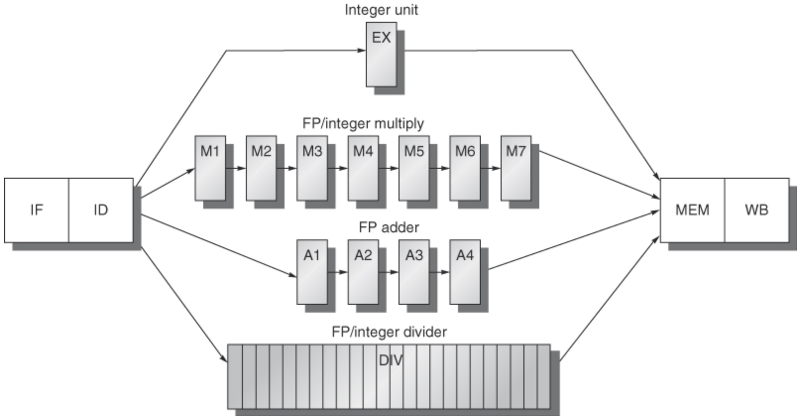


Figure C.35 A pipeline that supports multiple outstanding FP operations. The FP multiplier and adder are fully pipelined and have a depth of seven and four stages, respectively. The FP divider is not pipelined, but requires 24 clock cycles to complete. The latency in instructions between the issue of an FP operation and the use of the result of that operation without incurring a RAW stall

Figure: H&P Appendix C

Deeper Pipelines (2)

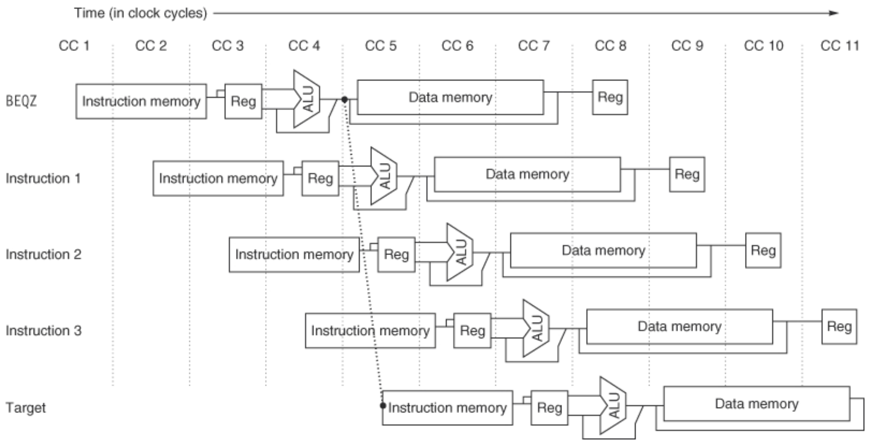
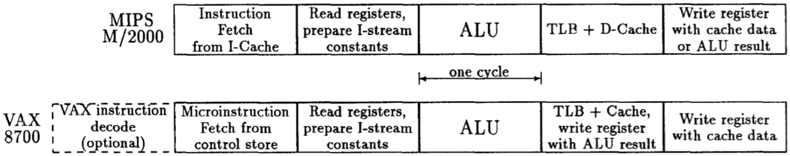


Figure C.44 The basic branch delay is 3 cycles, since the condition evaluation is performed during EX.

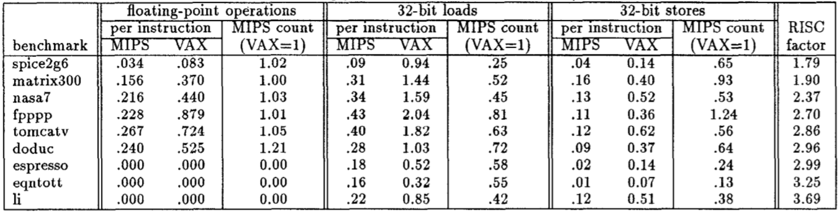
Figure: H&P Appendix C

Microcoded pipelined CPU


MIPS M/2000
Instruction Fetch from I-Cache
Read registers, prepare I-stream constants
ALU
TLB + D-Cache
Write register with cache data or ALU result
one cycle
VAX 8700
VAX instruction decode (optional)
Microinstruction Fetch from control store
Read registers, prepare I-stream constants
ALU
TLB + Cache, write register with ALU result
Write register with cache data

Less registers? (1)

Table 3: Floating-point operations and 32-bit loads/stores



	floating-point operations			32-bit loads			32-bit stores			
	per instruction		MIPS count	per instruction		MIPS count	per instruction		MIPS count	RISC
benchmark	MIPS	VAX	(VAX=1)	MIPS	VAX	(VAX=1)	MIPS	VAX	(VAX=1)	factor
spice2g6	.034	.083	1.02	.09	0.94	.25	.04	0.14	.65	1.79
matrix300	.156	.370	1.00	.31	1.44	.52	.16	0.40	.93	1.90
nasa7	.216	.440	1.03	.34	1.59	.45	.13	0.52	.53	2.37
fpppp	.228	.879	1.01	.43	2.04	.81	.11	0.36	1.24	2.70
tomcatv	.267	.724	1.05	.40	1.82	.63	.12	0.62	.56	2.86
doduc	.240	.525	1.21	.28	1.03	.72	.09	0.37	.64	2.96
espresso	.000	.000	0.00	.18	0.52	.58	.02	0.14	.24	2.99
eqntott	.000	.000	0.00	.16	0.32	.55	.01	0.07	.13	3.25
li	.000	.000	0.00	.22	0.85	.42	.12	0.51	.38	3.69

Less registers? + Seperate I-Cache?

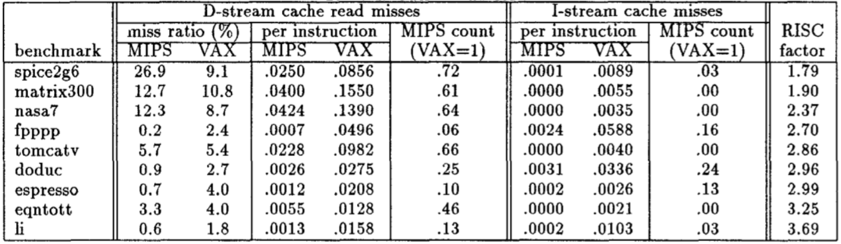


Table 4: Cache behavior

	D-stream cache read misses					I-stream cache misses			
	miss ratio (%)		per instruction		MIPS count	per instruction		MIPS count	RISC
benchmark	MIPS	VAX	MIPS	VAX	(VAX=1)	MIPS	VAX	(VAX=1)	factor
spice2g6	26.9	9.1	.0250	.0856	.72	.0001	.0089	.03	1.79
matrix300	12.7	10.8	.0400	.1550	.61	.0000	.0055	.00	1.90
nasa7	12.3	8.7	.0424	.1390	.64	.0000	.0035	.00	2.37
fpppp	0.2	2.4	.0007	.0496	.06	.0024	.0588	.16	2.70
tomcatv	5.7	5.4	.0228	.0982	.66	.0000	.0040	.00	2.86
doduc	0.9	2.7	.0026	.0275	.25	.0031	.0336	.24	2.96
espresso	0.7	4.0	.0012	.0208	.10	.0002	.0026	.13	2.99
eqntott	3.3	4.0	.0055	.0128	.46	.0000	.0021	.00	3.25
li	0.6	1.8	.0013	.0158	.13	.0002	.0103	.03	3.69

RISC factors

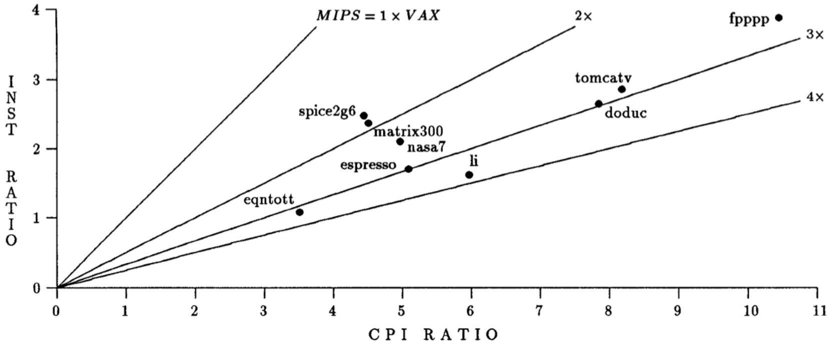


Figure 2: Instruction ratio versus CPI ratio. Lines of constant RISC factor are shown.

Factors favoring MIPS

operand specifier decoding — 1 cycle per on VAX

seperate floating point registers — seperate FPU

condition code RAW hazards

needless work by, e.g., CISC CALL/RET

filled delay slots

larger page size

larger range for brganches

Addressing modes on VAX

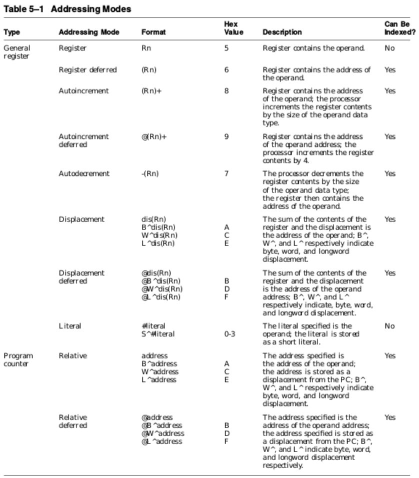


Table 5–1 Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Can Be Indexed?
General register	Register	Rn	5	Register contains the operand.	No
	Register deferred	(Rn)	6	Register contains the address of the operand.	Yes
	Autoincrement	(Rn)+	8	Register contains the address of the operand; the processor increments the register contents by the size of the operand data type.	Yes
	Autoincrement deferred	@(Rn)+	9	Register contains the address of the operand address; the processor increments the register contents by 4.	Yes
	Autodecrement	-(Rn)	7	The processor decrements the register contents by the size of the operand data type; the register then contains the address of the operand.	Yes
	Displacement	dis(Rn) B^dis(Rn) W^dis(Rn) L^dis(Rn)	 A C E	The sum of the contents of the register and the displacement is the address of the operand; B^, W^, and L^ respectively indicate byte, word, and longword displacement.	Yes
	Displacement deferred	@dis(Rn) @B^dis(Rn) @W^dis(Rn) @L^dis(Rn)	 B D F	The sum of the contents of the register and the displacement is the address of the operand address; B^, W^, and L^ respectively indicate, byte, word, and longword displacement.	Yes
	Literal	#literal S^#literal	 0-3	The literal specified is the operand; the literal is stored as a short literal.	No
Program counter	Relative	address B^address W^address L^address	 A C E	The address specified is the address of the operand; the address is stored as a displacement from the PC; B^, W^, and L^ respectively indicate byte, word, and longword displacement.	Yes
	Relative deferred	@address @B^address @W^address @L^address	 B D F	The address specified is the address of the operand address; the address specified is stored as a displacement from the PC; B^, W^, and L^ indicate byte, word, and longword displacement respectively.	Yes

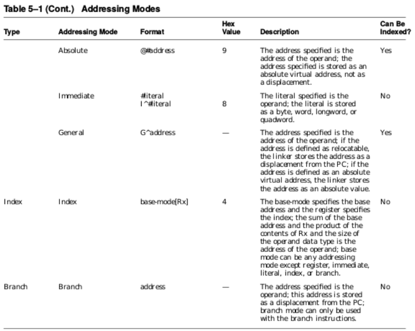


Table 5–1 (Cont.) Addressing Modes

Type	Addressing Mode	Format	Hex Value	Description	Can Be Indexed?
	Absolute	@#address	9	The address specified is the address of the operand; the address specified is stored as an absolute virtual address, not as a displacement.	Yes
	Immediate	#literal I^#literal	 8	The literal specified is the operand; the literal is stored as a byte, word, longword, or quadword.	No
	General	G^address	—	The address specified is the address of the operand; if the address is defined as relocatable, the linker stores the address as a displacement from the PC; if the address is defined as an absolute virtual address, the linker stores the address as an absolute value.	Yes
Index	Index	base-mode[Rx]	4	The base-mode specifies the base address and the register specifies the index; the sum of the base address and the product of the contents of Rx and the size of the operand data type is the address of the operand; base mode can be any addressing mode except register, immediate, literal, index, or branch.	No
Branch	Branch	address	—	The address specified is the operand; this address is stored as a displacement from the PC; branch mode can only be used with the branch instructions.	No

Addressing modes on VAX

```
ADDL3 @(R5)+[R6], @(R1)+[R2], @(R3)+[R4]
```

one instruction

six memory accesses, four register reads

three register writes

$$\begin{array}{lcl} \text{MEM[MEM[R5]}+\text{R6]} & \leftarrow & \text{MEM[MEM[R1]}+\text{R2]} \\ & + & \text{MEM[MEM[R3]}+\text{R4]} \\ \text{R1} & \leftarrow & \text{R1} + 4 \\ \text{R3} & \leftarrow & \text{R3} + 4 \\ \text{R5} & \leftarrow & \text{R5} + 4 \end{array}$$

ISA design

lots of non-technical factors

Notable RISC V decisions

modular ISA design

optional variable length encoding (code size)

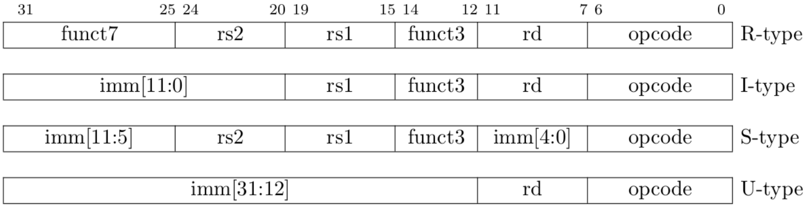
Justifications (1)

31 general-purpose registers + 0 register + pc

usually 32-bit instructions

> "it is impossible to encode a complete ISA with 16 registers in 16-bit instructions using a 3-address format. Although a 2-address format would be possible, it would increase instruction count and lower efficiency. … A larger number of integer registers also helps performance on high-performance code,…"
>
> "The optional compressed 16-bit instruction format mostly only accesses 8 registers"

Justifications (2)



31 25	24 20	19 15	14 12	11 7	6 0	
funct7	rs2	rs1	funct3	rd	opcode	R-type
imm[11:0]		rs1	funct3	rd	opcode	I-type
imm[11:5]	rs2	rs1	funct3	imm[4:0]	opcode	S-type
imm[31:12]				rd	opcode	U-type

Figure 2.2: RISC-V base instruction formats.

“Decoding register specifiers is usualy on the critical path … so the instruction format was chosen to keep all registers specifiers at the same position…”

Justifications (3)

no delay slots

no condition codes

“condition codes and branch delay slots, which complicate higher performance implementations”