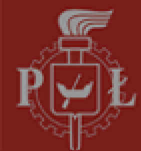


Post-RISC Architectures

DMCS

Performance Drivers

- Microelectronic technology advances are not determining the performance improvement
- Parallel processing at the instruction level (ILP – Instruction Level Parallelism) has become the key element of performance:
  - overlapped execution (pipelining)
  - true parallel execution (superscalar)
- Problem: programs are sequences of instructions that are highly dependent (inherent nature of computation)

Increasing the Parallelism

- Static methods: at compilation time
  - software optimization techniques
    - architecture independent
      - e.g. common expression elimination, code motion, ...
    - architecture dependent
      - pipeline scheduling, loop unrolling
- **Dynamic methods: at run-time**
  - dynamic scheduling
    - out-of-order execution
    - speculative execution

Identifying Opportunities

- Computer programs are build from basic sequences (of dependent instructions) executed in various configurations (loop, etc.)
- ILP must be exploited **across multiple basic block** (in one block independence is minimal)

- Example: Loop-Level-Parallelism

```
for (i=0; i<1000; i++)
   A[i] = A[i] + B[i]
```

→ to ILP by loop-unrolling (static or dynamic)

Pipeline Performance

- Ideal pipeline CPI = 1 (for single pipeline)
  - structural hazards
  - data hazards
  - control hazards

CPI = idealCPI + Struc.Stalls + DataStalls + Cont.Stalls

CPI is always > 1 in single pipeline implementation

IPC might be > 1 in superscalar architectures

DMCS

Data Hazards

- RAW (read-after-write): true data dependence
  - order must be preserved (correct data flow)
  - sets the limit to optimization techniques
- WAW (write-after-write): output dependence
  - order may be changed (with register renaming)
- WAR (write-after-read): anti-dependence
  - order may be changed (with register renaming)
- RAR (read-after-read): no conflicts

DMCS

Control Hazards

- Control dependence is not critical and can be violated during code optimization
- Instructions can be executed speculatively if this does not violate the program correctness

but it cannot also violate the following properties:

- exception behavior
- data flow

Exception Behavior Violation

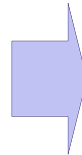
- Any change in program order cannot create a chance of rising any new exceptions

```
       ADD   R2,R3,R4
       BRZ   R2,skip
       LW    R1,(R2)
       ...
skip:
```

There is no true data dependence between BRZ and LW, so their order can be changed (providing that loading R1 will not be harmful)

(after dynamic scheduling)

The early load can cause unexpected memory protection exception



```
       ADD   R2,R3,R4
       LW    R1,(R2)
       BRZ   R2,skip
       ...
skip:
```

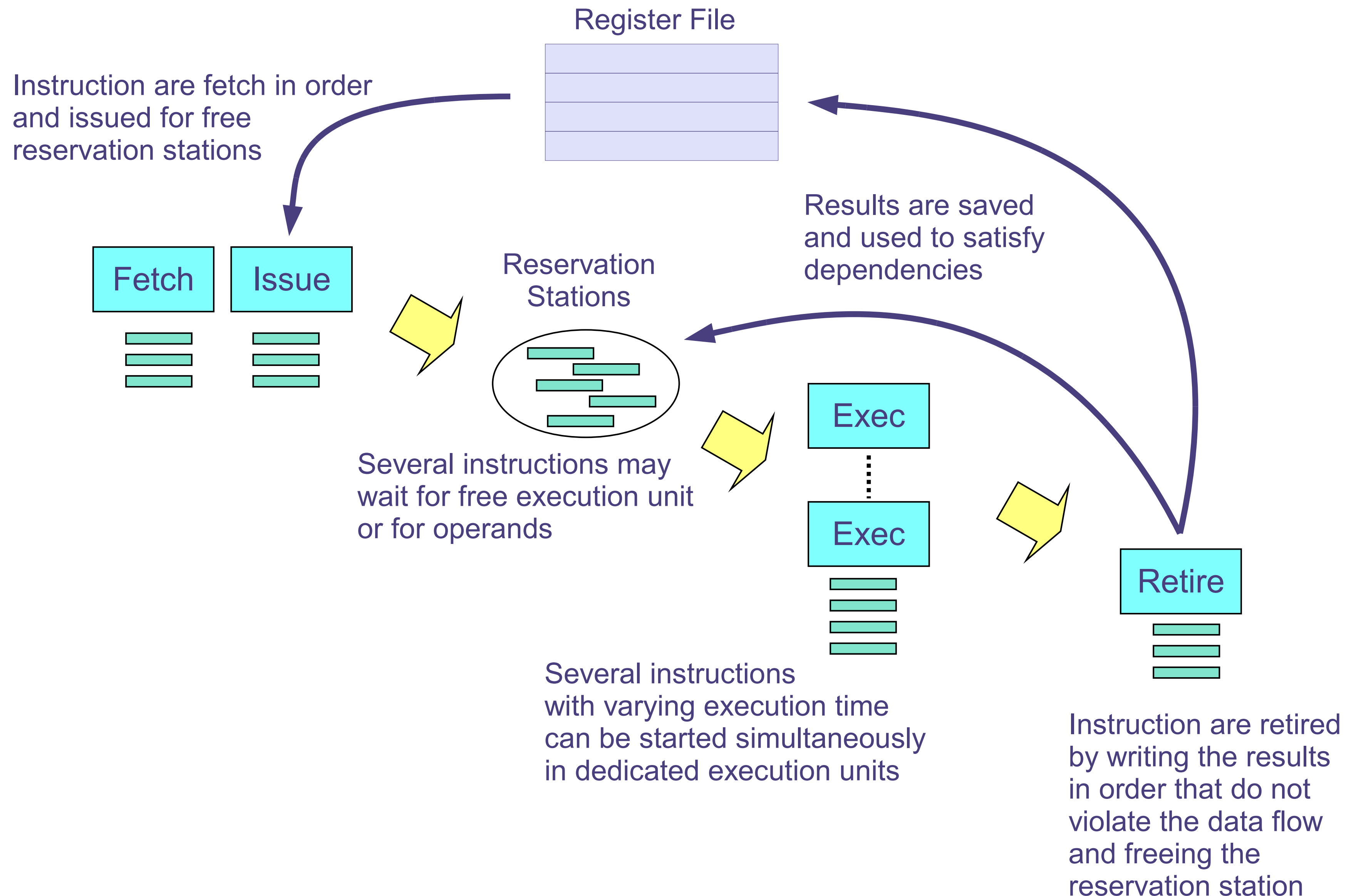
Dynamic Scheduling

- In single issue (rigid) pipeline the stalls can be reduced only by static (by compiler) optimization
- General pipeline optimization (for any code, including hard-to-optimize) can be obtained by dynamic (by hardware) scheduling at run time
- The dynamically scheduled pipeline is composed of the following stages:
  - Instruction Fetch
  - Instruction Issue
  - Instruction Execute (parallel with multiple units)
  - Instruction Retire

Dynamic Pipeline

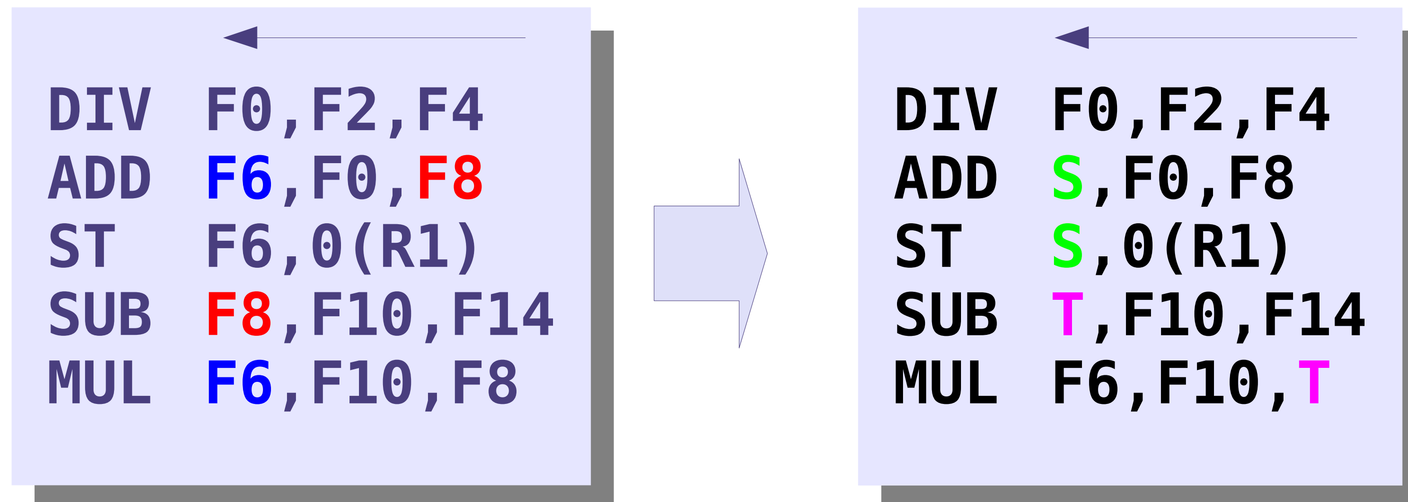
- Fetch – instructions are fetched in order and placed in the instruction queue
- Issue – instructions are read from the queue, decoded and placed in free reservation stations together with operands (or their dependencies)
- Execute – performed in parallel in separate units (out of order) for instructions with valid operands
- Retire – write results back to destination register and issue/exec instructions waiting for operands

Dynamic Pipeline



Register Renaming

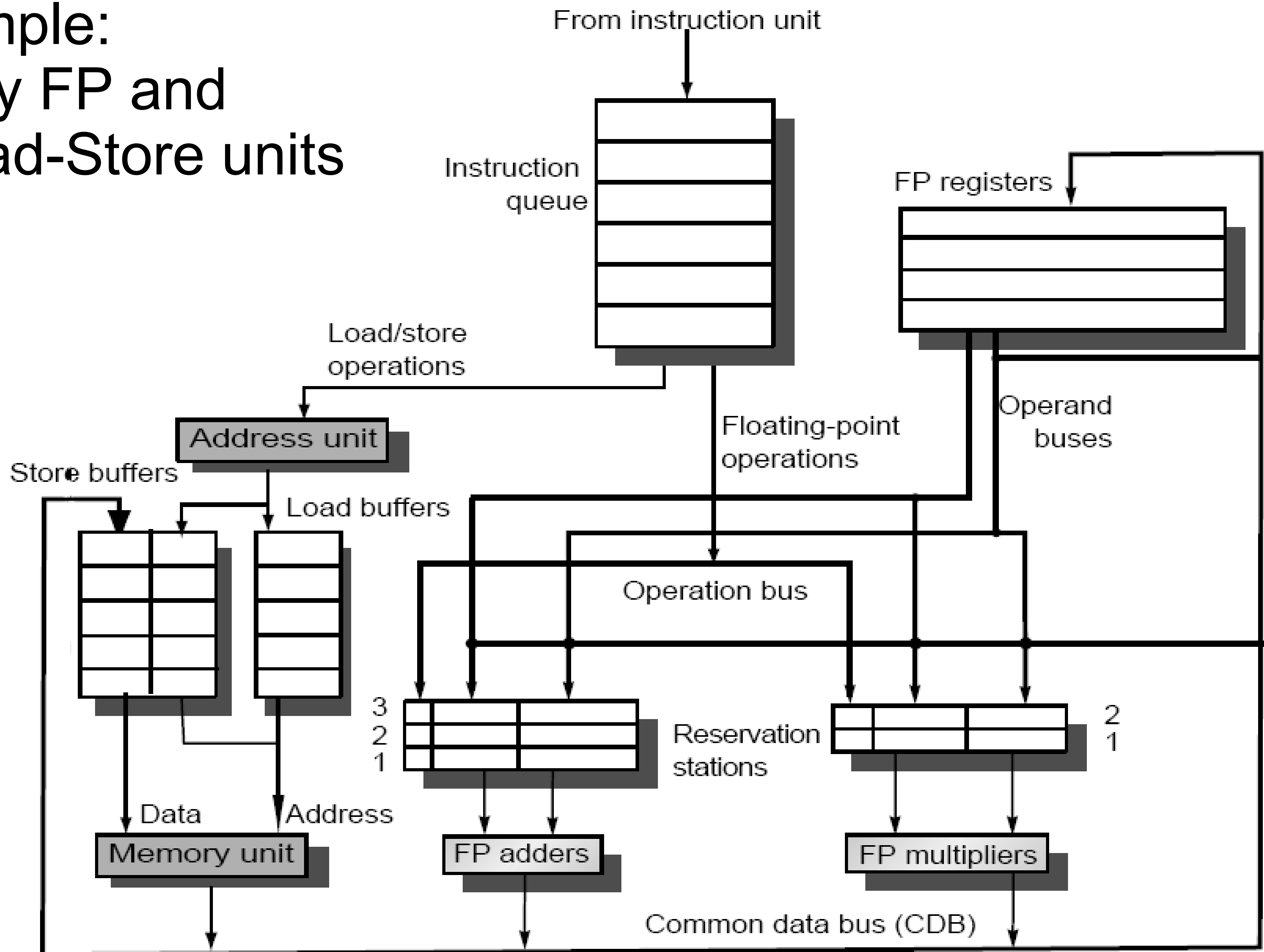
- WAR & WAW hazards can be eliminated by using separate registers (register renaming)
  - F8 – WAR (anti-dependence)
  - F6 – WAW (output-dependence)
- Additional, temporary registers are needed
- Temp. registers must be finally written back



Architektura komputerów, Informatyka sem. III
DMCS

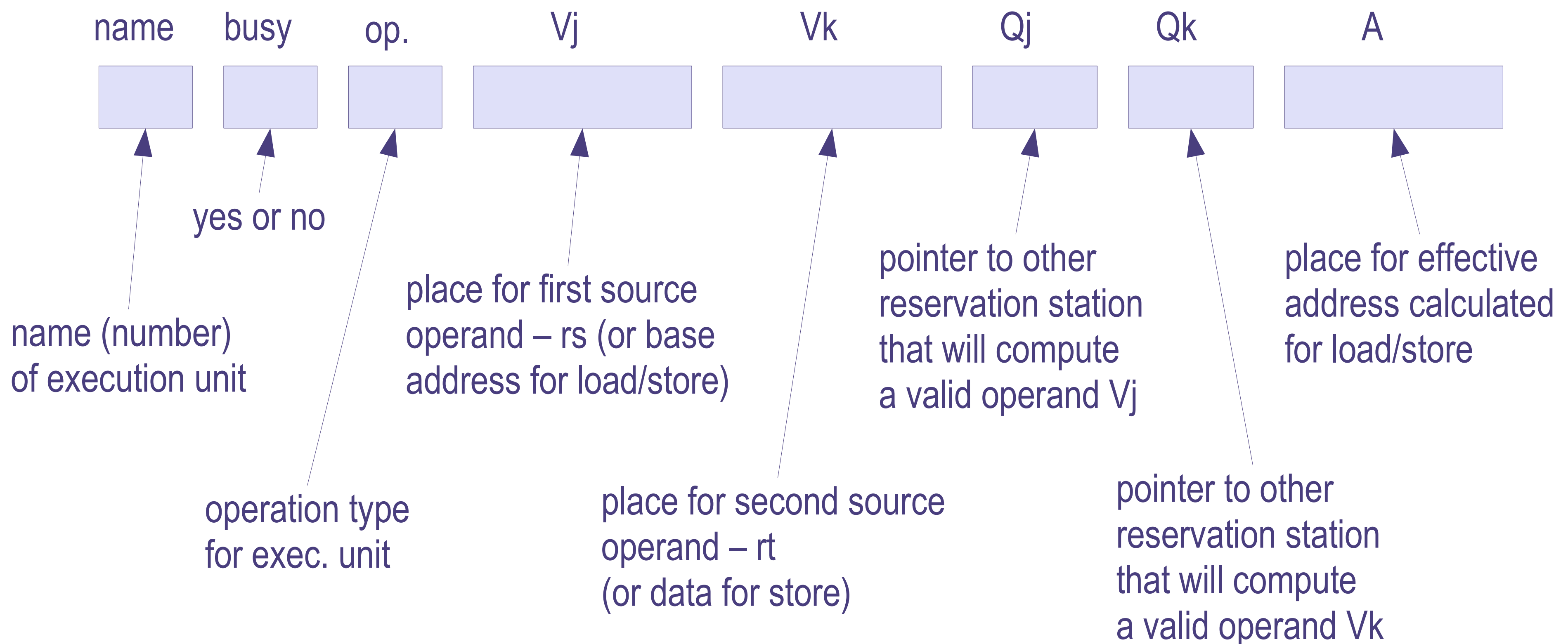
Dynamic Pipeline by Example

Example:
only FP and
Load-Store units



Reservation Stations

- Operands are stored directly (Vj & Vk) or registered to come from other res. station (Qj & Qk)
- Vj & Vk and Qj & Qk are mutually exclusive



Dynamic Scheduling Example 1

- Loads are independent
- MUL and SUB wait for loads
- MUL & SUB are independent
- DIV waits for MUL
- MUL & DIV take long time
- Last ADD need not wait for DIV

```
LD   F6,34(R2)
LD   F2,45(R3)
MUL  F0,F2,F4
SUB  F8,F2,F6
DIV  F10,F0,F6
ADD  F6,F8,F2
```

- All instructions can be issued without stalls, but some with operand-dependencies
- MUL & SUB can be executed in parallel
- ADD can be executed before DIV (WAR)

Example 1 cont.

Status after execution of two loads and retiring the first one

Instruction	Instruction status: Issue	Execute	Write result
L.D F6,34(R2)	√	√	√
L.D F2,45(R3)	√	√	
MUL.D F0,F2,F4	√		
SUB.D F8,F2,F6	√		
DIV.D F10,F0,F6	√		
ADD.D F6,F8,F2	√		

Reservation stations

Name	Busy	Op	Vj	Vk	Qj	Qk	A
Load1	no						
Load2	yes	Load					45+Regs[R3]
Add1	yes	SUB		Mem[34+Regs[R2]]	Load2		
Add2	yes	ADD			Add1	Load2	
Add3	no						
Mult1	yes	MUL		Regs[F4]	Load2		
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

Register status

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
Qi	Mult1	Load2		Add2	Add1	Mult2			

Each (normal) register has a status field, that is points to the reservation station holding the instruction that will finally provide the most up-to-date result

Example 1 cont.

Status after execution of all instructions except the floating-point MUL and DIV

Instruction	Issue	Execute	Write result
	Instruction status		
L.D F6,34(R2)	√	√	√
L.D F2,45(R3)	√	√	√
MUL.D F0,F2,F4	√	√	
SUB.D F8,F2,F6	√	√	√
DIV.D F10,F0,F6	√		
ADD.D F6,F8,F2	√	√	√

Name	Busy	Op	Vj	Vk	Qj	Qk	A
	Reservation stations						
Load1	no						
Load2	no						
Add1	no						
Add2	no						
Add3	no						
Mult1	yes	MUL	Mem[45+Regs[R3]]	Regs[F4]			
Mult2	yes	DIV		Mem[34+Regs[R2]]	Mult1		

Field	F0	F2	F4	F6	F8	F10	F12	...	F30
	Register status								
Qi	Mult1					Mult2			

Each (normal) register has a status field, that is points to the reservation station holding the instruction that will finally provide the most up-to-date result

DMCS

Additional Issues

- Exceptions rising with out-of-order execution
  - exceptions must be registered but not processed until all the preceding branches are retired
- Load/Store conflicts
  - issuing the load/store instructions must be preceded by examination of already issued load/store instructions for address collision during:
    - load-after-store
    - store-after-load
    - store-after-store

Dynamic Scheduling

- Pros:
  - simple compliers / hard-to-schedule code
  - relatively small number of registers
  - aggressive approach to performance
- Cons:
  - very complex hardware / cost
  - optimizing compliers for architecture
  - complication to branch prediction and exception rising

PŁ
DMCS

Reducing Control Hazard

- Conditional branches in programs are unavoidable!
- Reduction of branch hazard is important in single-issue pipeline (SIP), but is **crucial for multiple-issue pipeline** (MIP) (dynamic scheduling, IPC>1)
- According to Amdahl's Law, control hazards will lower the benefits from dynamic scheduling and static code optimization
- Simple static schemes: are not sufficient in MIP
  - delayed branch
  - predict not taken (with pipeline flush)
- Correct dynamic branch prediction is necessary!

Dynamic Branch Prediction

- Prediction depends on the run-time behavior
- The branch behavior may change (is dynamic)
- Static prediction is impossible
- Dynamic prediction must be handled by hardware, at run-time
- Prediction accuracy is important
- For MIP processors, cooperation between dynamic scheduling and branch prediction introduces another level of complexity (hardware speculation)

DMCS

Prediction Effectiveness

- Control hazard reduction is a two-fold problem
  - prediction effectiveness
    - the ratio of correct predictions over the total branch number
    - complexity (cost/speed) of prediction hardware
  - time to prediction validation
    - simple vs advanced conditional branches
    - complexity of condition evaluation hardware
    - cost of pipeline flush
    - misprediction recovering strategies
- Total impact on MIP performance and complexity
  - hardware cost/size trade-offs

PŁ DMCS

Branch History Table (BHT)

- Small dedicated cache accessed during IF-stage, with entries corresponding only to branches
- Contains the target address left by the last taken branch instruction (validated)
- Each entry has a prediction indicator
  - last branch taken (T) or not (NT) - simplest scheme
- Prediction is just a hint, it might be incorrect
  - due to the evaluation of current branch condition
- BHT is updated after branch validation
- In SIP validation & pipeline flush is simple

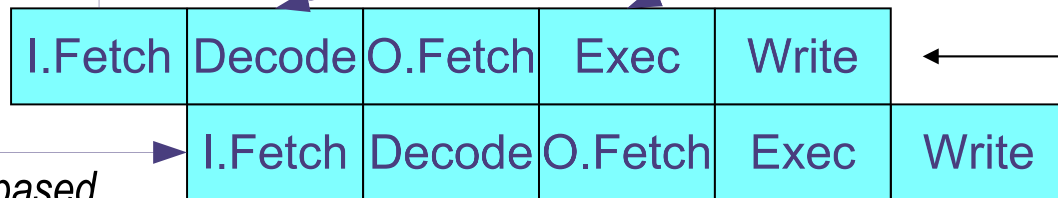
Branch History Table

- For SIP performance improvement may be small
- For MIP – the key element to performance

Branch History Table
(Branch Target Buffer)

Prediction	Target Address	Instruction Address
T		

Validation
fast (simple conditions, no data hazards allowed)
or
late (complex conditions, data hazards allowed



Flush ?

Branch

The guess based on branch table

Single Issue Pipeline (SIP)

Prediction Indicator

- One-bit scheme: simplest, with obvious limitation
  - first (NT) and last (T) loop misprediction
  - e.g 10-cycle loop → 80% prediction accuracy
- Two-bit scheme: an easy improvement
  - only last (T) loop misprediction
  - e.g 10-cycle loop → 90% prediction accuracy
- General n-bit predictors
  - n-bit for-and-back counters
  - loop taken or not, according to a threshold value
  - 2-bit scheme is the best (effectiveness/complexity)

DMCS

2-bit Prediction Scheme

Taken

Not taken

Predict taken
11

Predict taken
10

Taken

Taken

Not taken

Not taken

Predict not taken
01

Predict not taken
00

Taken

Not taken

Branch Prediction Results

4k-entry
2-bit BHT

Prediction accuracy is lower for integer benchmarks (complex data-flow algorithms) and higher for FP benchmarks (scientific loop-oriented calculations)

SPEC89 benchmarks

Benchmark	Frequency of mispredictions
nasa7	1%
matrix300	0%
tomcatv	1%
doduc	5%
spice	9%
fpppp	9%
gcc	12%
espresso	5%
eqntott	18%
li	10%

Axis: 0% 2% 4% 6% 8% 10% 12% 14% 16% 18%

Frequency of mispredictions

Prediction accuracy (frequency) influence performance together with branch frequency !!!

100% − Prediction accuracy

Hennessy, Patterson, Computer.Architecture.-.A.Quantitative.Approach

Prediction Buffer Size

- How large BHT is large enough?
- Several k-entries seems to be satisfactory

SPEC89 benchmarks

Benchmark	4096 entries: 2 bits per entry	Unlimited entries: 2 bits per entry
nasa7	1%	0%
matrix300	0%	0%
tomcatv	1%	0%
doduc	5%	5%
spice	9%	9%
fpppp	9%	9%
gcc	12%	11%
espresso	5%	5%
eqntott	18%	18%
li	10%	10%

0% 2% 4% 6% 8% 10% 12% 14% 16% 18%

Frequency of mispredictions

100% − Prediction accuracy

Hennessy, Patterson, Computer.Architecture.-.A.Quantitative.Approach

Prediction Schemes

- Simple scheme – recent branch behavior
- Advanced scheme
  - recent branch behavior of current & other branches
- Branch predictors may be **correlated**
  - e.g. if first and second are NT,the third is always T

DMCS

Correlating Branch Predictors

- Example:

```
if (d==0)          //b1 branch
          d=1;
if (d==1)          //b2 branch
          {…}
```

```
    BNEZ   R1,L1      ;branch b1(d!=0)
    ADDI   R1,R0,#1   ;d==0, so d=1
L1: ADDI   R3,R1,#-1
    BNEZ   R3,L2      ;branch b2(d!=1)
...
L2:
```

- Possible execution sequences
  - strong correlation: last b1 NT → next b2 NT

Initial value of d	d==0?	b1	Value of d before b2	d==1?	b2
0	yes	not taken	1	yes	not taken
1	no	taken	1	yes	not taken
2	no	taken	2	no	taken

cont.

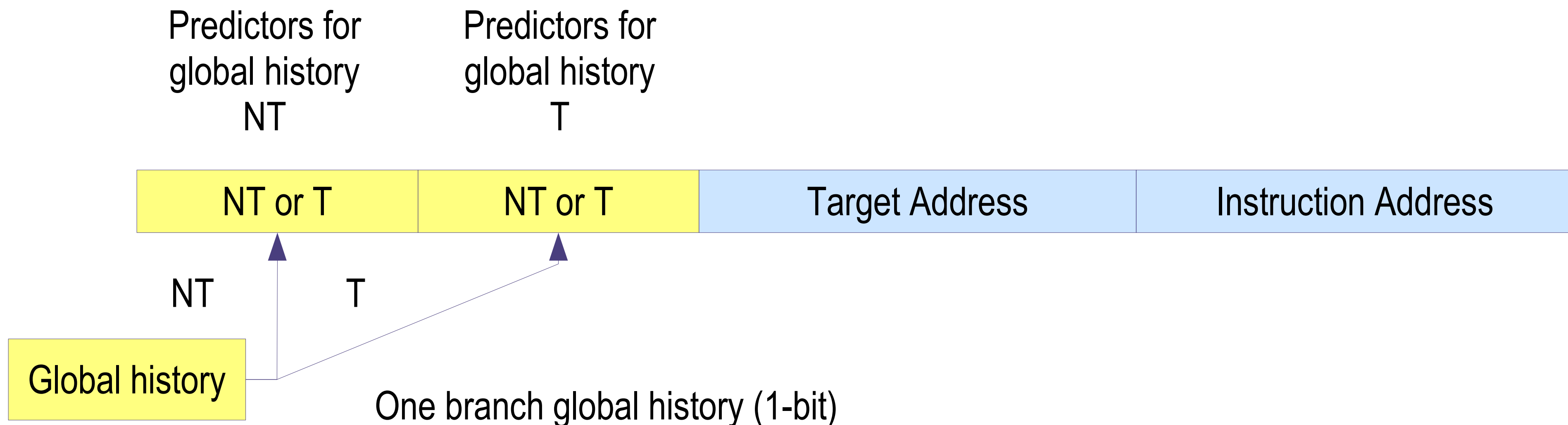
- Example for alternating values of d (2,0,2,0,...)
- Simple 1-bit predictor will fail in all predictions!

```
if (d==0)          //b1 branch
          d=1;
if (d==1)          //b2 branch
          {…}
```

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT
2	NT	T	T	NT	T	T
0	T	NT	NT	T	NT	NT

Correlating Predictors

- The behavior of (any!) last executed branch (i.e. 1-bit global history) is considered – T or NT
- Each BHT entry is accompanied by two predictor fields
- The prediction is made by querying the predictor pointed by the global history



Correlating Predictors

- All, except first, predictions are correct

```
if (d==0)          //b1 branch
        d=1;
if (d==1)          //b2 branch
        {…}
```

(let's analyze the behavior of b2 branch only, with correlation to b1)

d=?	b1 prediction	b1 action	New b1 prediction	b2 prediction	b2 action	New b2 prediction
2	NT/NT	T	T/NT	NT/**NT**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T
2	**T**/NT	T	T/NT	NT/**T**	T	NT/T
0	T/**NT**	NT	T/NT	**NT**/T	NT	NT/T

Global history for branch b2 prediction

Predictions for b2 made with the use of correlation with the b1 branch

(1,1) Predictor

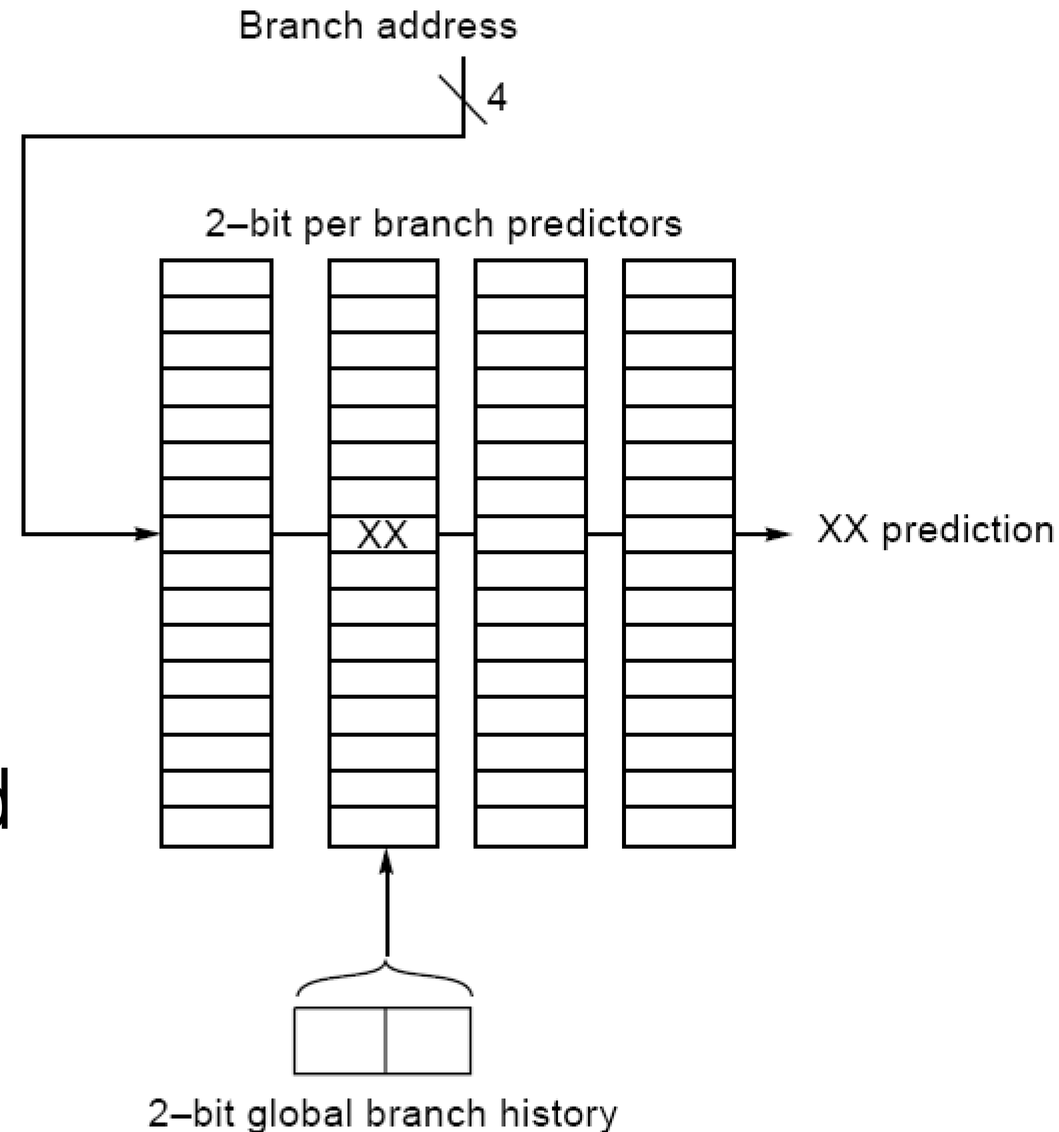
- The presented example of correlating predictor is called (1,1) scheme
  - 1 – global history consist of (one) last branch
  - 1 – each predictor uses 1-bit prediction strategy
- e.g. the (2,2) predictor:
  - 2 – global history consist of 2 recent branches (4 alternative predictor fields needed!)
  - 2 – each predictor uses 2-bit prediction strategy
- e.g. the (0,1) predictor - simplest 1-bit scheme
- e.g. the (0,2) predictor - 2-bit scheme, no corr.

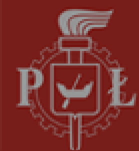
(m,n) Predictors

- General case of (m,n) would mean:
  - m – global history consist of m-recently executed branches (2^m alternative predictor fields needed)
  - n – each predictor uses n-bit prediction strategy
- BHT predictors memory demand:
  - $2^m * n *$ BHT_entries [in bits]
  - e.g. (2,2) 4k BHT $\longrightarrow$ 4*2*4096 b = 2^{15} kb = 32 kb (not much compared to the total BHT size)

(2,2) Predictor Implementation

- Global history can be implemented as shift register, with bits 0(NT), and 1(T)
- After the selection, the prediction is made
- After branch validation the predictor is updated
- Hardware is not big and not too complex!





Prediction Schemes Comparison

- Comparison of equal-size of BHT predictor bits: (0,2) 4k = (2,2) 1k
- Correlating predictors perform much better

Frequency of mispredictions — SPEC89 benchmarks

Benchmark	4096 entries: 2 bits per entry	Unlimited entries: 2 bits per entry	1024 entries (2,2)
nasa7	1%	0%	1%
matrix300	0%	0%	0%
tomcatv	1%	0%	1%
doduc	5%	5%	5%
spice	9%	9%	5%
fpppp	9%	9%	5%
gcc	12%	11%	11%
espresso	5%	5%	4%
eqntott	18%	18%	6%
li	10%	10%	5%

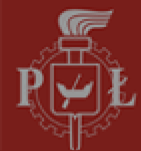
Hennessy, Patterson, Computer.Architecture.-.A.Quantitative.Approach

DMCS

Predictors Overview

- (0,2) and (2,2) schemes are most popular
- Multiple BHT entries vs Global History length
  - e.g. (12,2) with few entries performs surprisingly well
- Global (correlating) vs local prediction schemes
  - local (0,2) perform better than global (2,2) in some cases
- Tournament Predictors - multilevel predictors
  - several levels of predictors (local and global) and adaptively using most appropriate scheme

DMCS



Predictors Scheme Comparison

Misprediction Rate

8%
7%
6%
5%
4%
3%
2%
1%
0%

Local 2-Bit Predictor

Correlating Predictor

Tournament Predictor

SPEC89

0 32 64 96 128 160 192 224 256 288 320 352 384 416 448 480 512

Total predictor size

DMCS

Hardware-Based Speculation

- High performance results from:
  - exploiting instruction parallelism
    - superscalar (or VLIW,EPIC) architectures
  - reducing data hazards
    - static scheduling
  - overcoming data hazards
    - dynamic scheduling
  - reducing control hazards
    - advanced branch prediction
  - **overcoming control hazards**
    - **speculative execution**

Speculative Execution

- Conditional branches cannot halt the execution
- When the branch **prediction** is known, the following instructions will be executed **speculatively**
- Until the branch is validated, the results of the speculatively executed instructions must wait in a buffer
- If the prediction is found correct, the waiting results are allowed to modify the registers and memory
- If the prediction is found incorrect, all the waiting results are flushed

Pipeline with Speculation

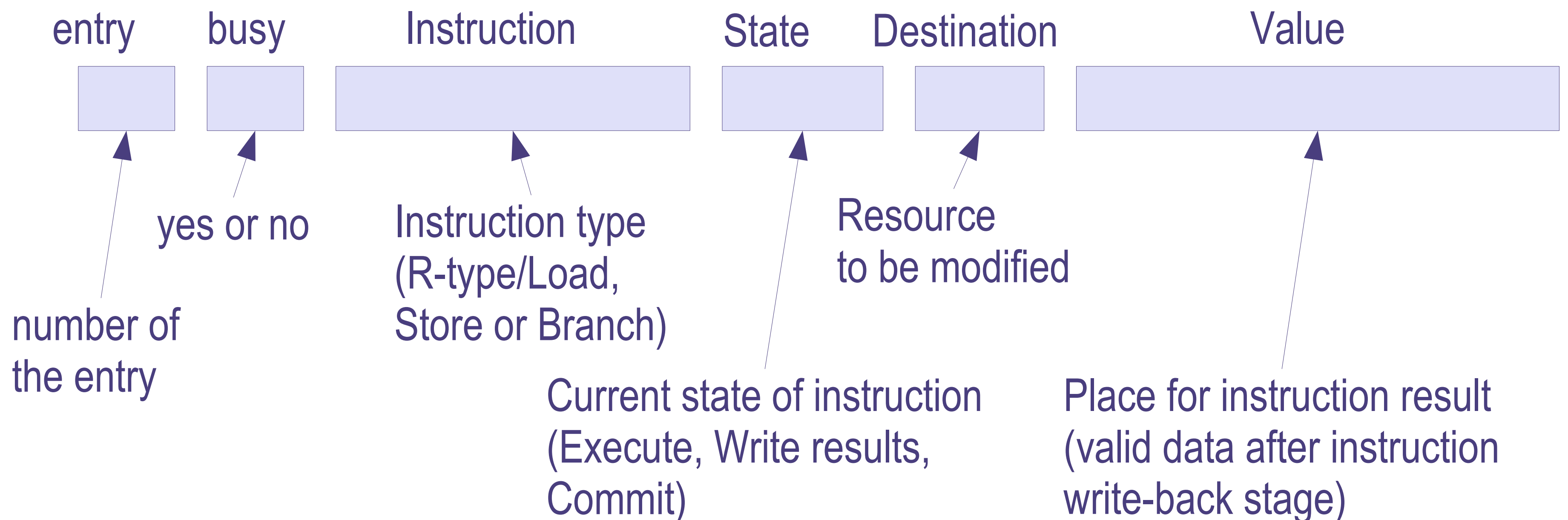
- Fetch
  - multiple instructions, with advanced branch prediction
- Issue
  - multiple instructions, resolving data hazards
- Execute
  - in multiple execution units and variable time
- Write back
  - broadcast & store results in instruction reorder buffer (ROB)
- Commit
  - finalize instructions from ROB – modify regs/memory

Pipeline with Speculation

- Fetch, Issue – instructions in order
  - instruction can be issued when there are both: free reservation station and free ROB entry
- Execute, Write-back – out of order
- Commit – instructions in order
  - entries are allocated (and numbered) for issued instructions in order
  - instructions stay in ROB until their result is ready
  - instructions are committed in order – conditional branches will be validated in due order
  - commit stage is fast, since the results are ready – multiple instruction can be committed

Reorder Buffer

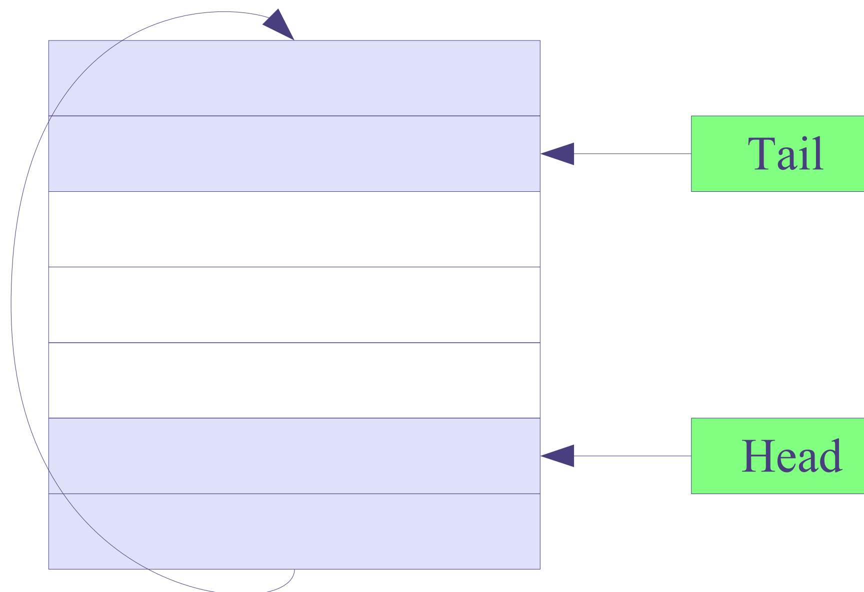
- ROB entries are used to provide operands for instructions in issue stage (apart from normal registers)
- Reservation stations and broadcasts can use ROB entry numbers to solve dependencies (instead of reservation stations numbers)



Architektura komputerów, Informatyka sem. III

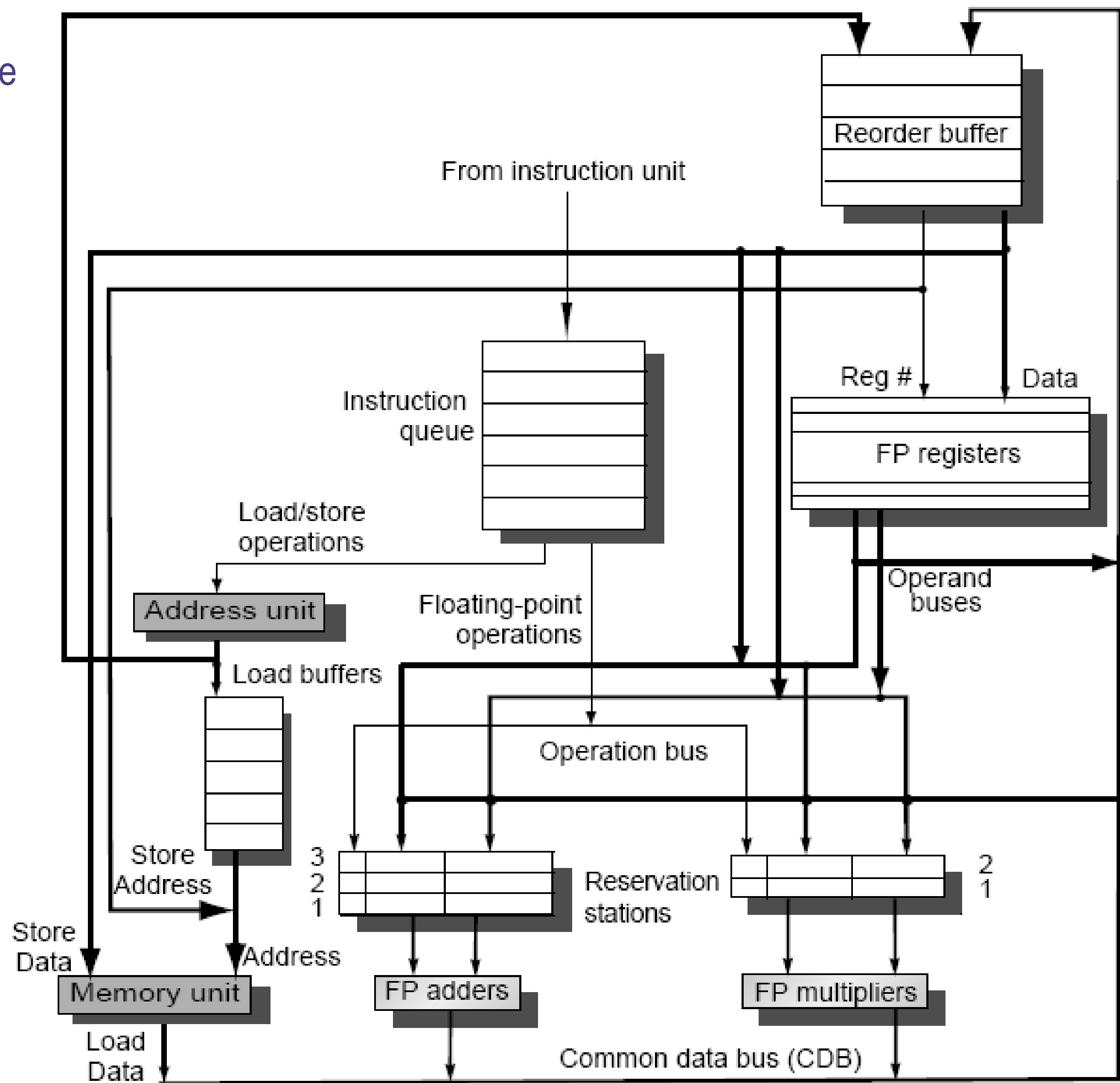
Reorder Buffer

- ROB is used in cyclical mode
  - there is always a head (first) entry
  - there is always a tail (last) entry
  - finding available (free) entries is easy



Pipeline with ROB

Example:
- only FP and Load-Store units



Instruction Treatment

- R-type & Load
  - issued to ROB & reservation stations, compute results in execution stage, update ROB in write-back,
    but update destination register only in commit stage
- Store
  - issued to ROB & address unit, waits in ROB for result to be stored, but modifies memory only in commit stage
- Conditional branch
  - issued to ROB & reservation stations, compute condition in execution stage, update ROB in write-back
    and gives flush/no flush information when committed

Speculative Exec. - Example

- Assume branch was correctly predicted & taken
- Several loop cycles can be issued and executed, waiting to be committed in order

```
loop:
        L.D      F0,0(R1)
        MUL.D    F4,F0,F2
        S.D      F4,0(R1)
        DADDI    R1,R1,#-8
        BNE      R1,R2,loop
```


DMCS

Speculative Exec. - Example

Snapshot at the moment of committing the first store

	Reorder buffer				
Entry	**Busy**	**Instruction**	**State**	**Destination**	**Value**
1	no	`L.D F0,0(R1)`	Commit	F0	Mem[0+Regs[R1]]
2	no	`MUL.D F4,F0,F2`	Commit	F4	#1 x Regs[F2]
3	yes	`S.D F4,0(R1)`	Write result	0+Regs[R1]	#2
4	yes	`DADDIU R1,R1,#-8`	Write result	R1	Regs[R1]–8
5	yes	`BNE R1,R2,Loop`	Write result		
6	yes	`L.D F0,0(R1)`	Write result	F0	Mem[#4]
7	yes	`MUL.D F4,F0,F2`	Write result	F4	#6 x Regs[F2]
8	yes	`S.D F4,0(R1)`	Write result	0+#4	#7
9	yes	`DADDIU R1,R1,#-8`	Write result	R1	#4 – 8
10	yes	`BNE R1,R2,Loop`	Write result		

	FP register status								
Field	**F0**	**F1**	**F2**	**F3**	**F4**	**F5**	**F6**	**F7**	**F8**
Reorder #	6				7				
Busy	yes	no	no	no	yes	no	no	...	no

DMCS

Commit Stage

- Speculative pipeline implements the dynamic scheduling
- All resource modifications are postponed until the instructions are committed in order
- Exceptions are raised only by instructions when committed – correct program behavior
- Mispredicted branches flush all the following ROB entries (and free the corresponding reservation stations)
- Misprediction penalty is very high with speculation – correct prediction is crucial to performance

MIP with Speculation

- Update of multiple ROB entries
- Monitoring of multiple CDB
- Performing multiple instruction commit

  - Example: 2-issue

    separate address calculation, integer ALU and branch condition evaluation units

```
Loop:   LW      R2,0(R1)    ; R2=array element
        DADDIU  R2,R2,#     ; increment R2
        SW      0(R1),R2    ; store result
        DADDIU  R1,R1,#4    ; increment pointer
        BNE     R2,R3,LOOP  ; branch if last element!=0
```

MIP without Speculation

Iter. #	Instructions	Issues at clock cycle #	Executes at clock cycle #	Memory access at clock cycle #	Write CDB at clock cycle #	Comment
1	LW R2,0(R1)	1	2	3	4	First issue
1	DADDIU R2,R2,#1	1	5		6	Wait for LW
1	SW 0(R1),R2	2	3	7		Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	Execute directly
1	BNE R2,R3,LOOP	3	7			Wait for DADDIU
2	LW R2,0(R1)	4	8	9	10	Wait for BNE
2	DADDIU R2,R2,#1	4	11		12	Wait for LW
2	SW 0(R1),R2	5	9	13		Wait for DADDIU
2	DADDIU R1,R1,#4	5	8		9	Wait for BNE
2	BNE R2,R3,LOOP	6	13			Wait for DADDIU
3	LW R2,0(R1)	7	14	15	16	Wait for BNE
3	DADDIU R2,R2,#1	7	17		18	Wait for LW
3	SW 0(R1),R2	8	19	20		Wait for DADDIU
3	DADDIU R1,R1,#4	8	14		15	Wait for BNE
3	BNZ R2,R3,LOOP	9	19			Wait for DADDIU

MIP **with** Speculation

Iter. #	Instructions	Issues at clock #	Executes at clock #	Read access at clock #	Write CDB at clock #	Com-mits at clock #	Comment
1	LW R2,0(R1)	1	2	3	4	5	First issue
1	DADDIU R2,R2,#1	1	5		6	7	Wait for LW
1	SW 0(R1),R2	2	3			7	Wait for DADDIU
1	DADDIU R1,R1,#4	2	3		4	8	Commit in order
1	BNE R2,R3,LOOP	3	7			8	Wait for ADDDI
2	LW R2,0(R1)	4	5	6	7	9	No execute delay
2	DADDIU R2,R2,#1	4	8		9	10	Wait for LW
2	SW 0(R1),R2	5	6			10	Wait for DADDIU
2	DADDIU R1,R1,#4	5	6		7	11	Commit in order
2	BNE R2,R3,LOOP	6	10			11	Wait for DADDIU
3	LW R2,0(R1)	7	8	9	10	12	Earliest possible
3	DADDIU R2,R2,#1	7	11		12	13	Wait for LW
3	SW 0(R1),R2	8	9			13	Wait for DADDIU
3	DADDIU R1,R1,#4	8	9		10	14	Executes earlier
3	BNE R2,R3,LOOP	9	11			14	Wait for DADDIU

DMCS