COMPLEXITY EFFECTIVE SUPERSCALAR PROCESSORS

- **Part-I:**
  - **Objective:** *Characterizing Complexity at architecture level*
  - Baseline Architecture
  - Sources of Complexity
    - µArchitecture components such that ILP ↗ → complexity ↗
    - Models for quantifying component delays
- **Part-II:**
  - **Objective:** *Propose a Complexity-Effective µArchitecture*
    - High IPC & High Clock Rate

CHARACTERIZING COMPLEXITY

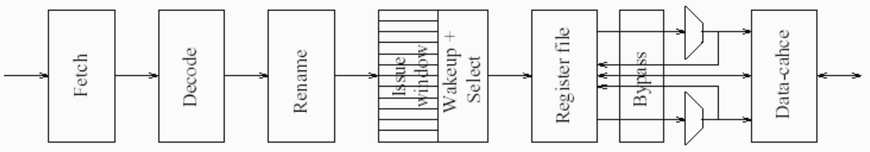
- **Complexity:**

  Delay through critical path
- Baseline Architecture ◀
- Defining Critical Structures
- Method for Quantifying Complexity
- Analysis of Critical Structures

<Mostly from [2]>

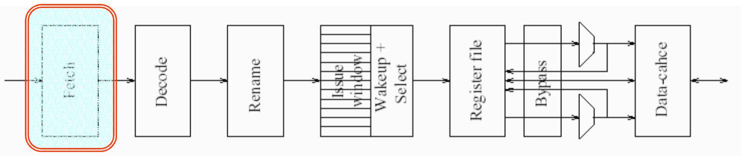
BASELINE ARCHITECTURE

- Superscalar, o-o-o execute, in order complete
  - MIPS R10000, DEC Alpha 21264



FETCH	DECODE	RENAME INSERT	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

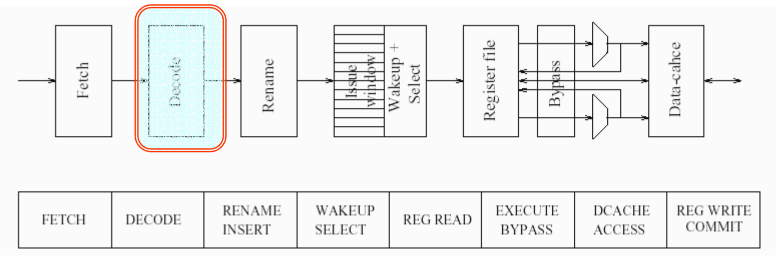
BASELINE ARCHITECTURE



FETCH	DECODE	RENAME INSERT	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

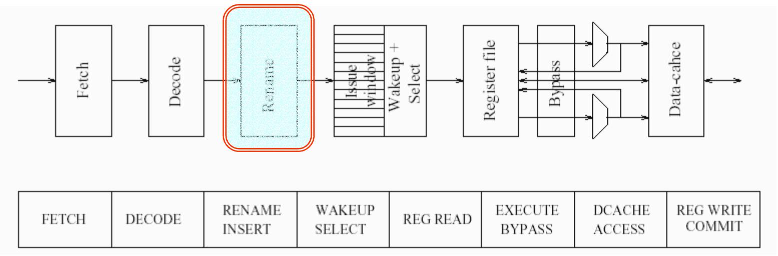
- Fetch:
  - Read Fetch-Width Instr-s/clk from I$
  - Predict Encountered Branches
  - Send to decoder

BASELINE ARCHITECTURE



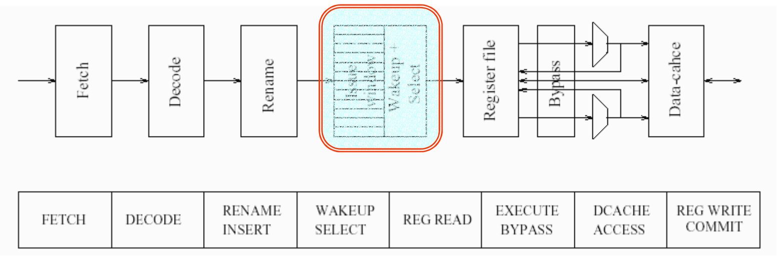
- Decode:
  - Decode instructions into op|subop|imm.|operands|etc.

BASELINE ARCHITECTURE



- Rename:
  - Rename the logical operand registers
    - Eliminate WAR and WAW
  - Logical register → physical register
  - ***Dispatch*** to Issue Window (Instruction Pool)

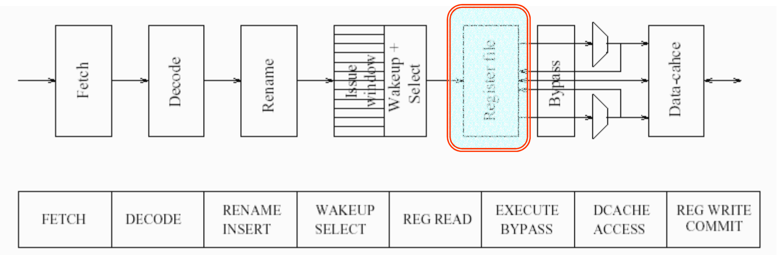
BASELINE ARCHITECTURE



❖Issue Window & Wakeup-Select Logic:

- Wait for source operands to be ready
- Issue instructions to exec. Units if ➔
  Source operands ready & functional unit available
- Fetch operands from Regfile – or bypass

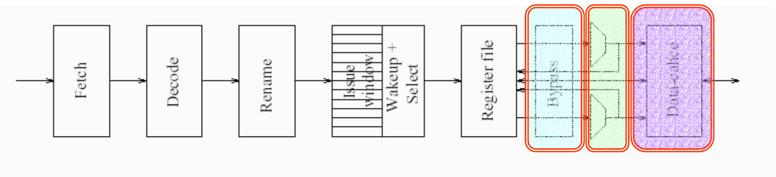
BASELINE ARCHITECTURE



❖Register File:

- Hold the physical registers
- Send the operands of currently issued instructions to exec. Units – or bypass

BASELINE ARCHITECTURE

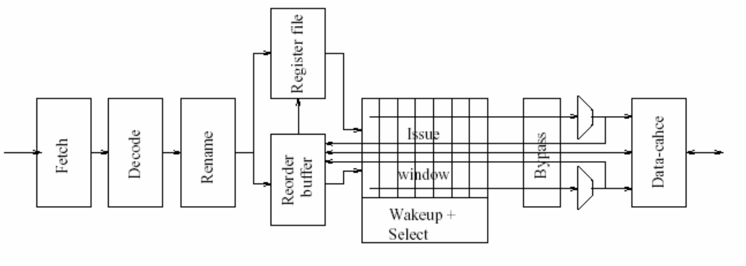


FETCH	DECODE	RENAME INSERT	WAKEUP SELECT	REG READ	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

- ❖Rest of Pipeline:
  - Bypass Logic
  - Execution Units
  - Data Cache

OTHER ARCHITECTURES

- ❖ Reservation Station Model:



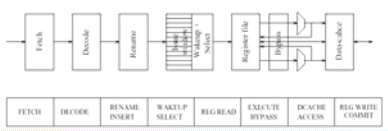
FETCH	DECODE	RENAME	REG READ ROB READ INSERT	WAKEUP SELECT	EXECUTE BYPASS	DCACHE ACCESS	REG WRITE COMMIT

- ❖ Intel P6, PowerPC 604

Baseline vs. Reservation Station

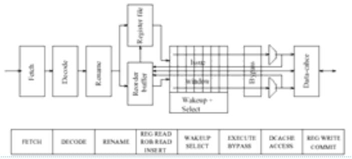
- Two Major Differences:

Baseline Model:



- All reg. values reside in physical reg-file
- Only tags of operands broadcast to window
  - Values go to physical reg-file

Res. Station Model:



- Reorder buffer holds speculative values; reg-file holds commited values
- Completing intsr-s broadcast operand values to reservation station
  - Issued instr-s read values from res. station

CHARACTERIZING COMPLEXITY

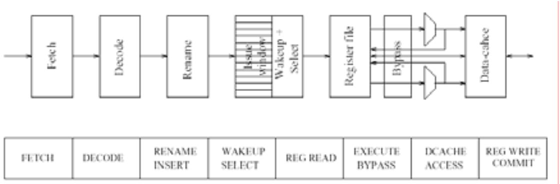
- **Complexity:**

  Delay through critical path
- Baseline Architecture
- Defining Critical Structures ◀
- Method for Quantifying Complexity
- Analysis of Critical Structures

<Mostly from [2]>

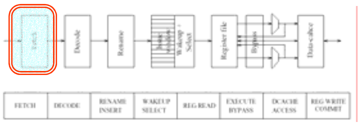
CRITICAL STRUCTURES

- Structures with Delay α
  Issue Width(*IW*) | Issue Window(*WinSize*)
- Dispatch & Issue related structures
- Structures that broadcast over long wires
- <u>Candidate Structures:</u>
  - Instruction Fetch Logic
  - Rename Logic
  - Wakeup Logic
  - Select Logic
  - Register File
  - Bypass Logic
  - Caches



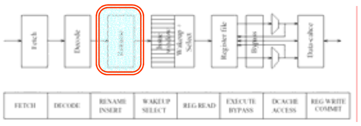
Instruction Fetch Logic

- Complexity
  α Dispatch/Issue Width



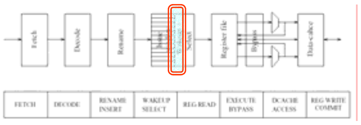
- As instr. Issue width ↗
  → Predict Multiple branches
- Non contiguous cache blocks need to be fetched and compacted
- Logic Described in [5]
- Delay Models to be developed

Register Rename Logic



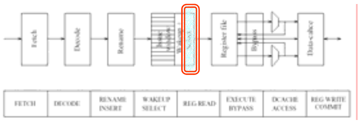
- ***Map Table:*** Logical to Physical Register Mapping
  - IW ↗ ➔ Number of map table ports ↗
- ***Dependence Check Logic:*** Detects true dependences within current rename group
  - IW ↗ ➔ Depth of Dep. Check Logic↗
- Delay α Issue Width

Wakeup Logic



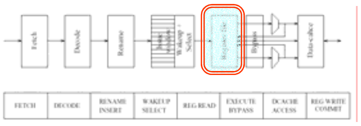
- Part of Issue Window
- 'Wake up' Instr-s when source operands ready
- When an instr. Issued, its result register tag broadcast to all instructions in issue window
  - WinSize ↗ ➔ Broadcast Fanout ↗ & Wire Length ↗
  - IW ↗ ➔ Size of each window entry ↗
- Delay α Issue Width & Window Size

Selection Logic



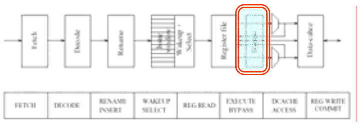
- ❖Part of Issue Window
- ❖Select Instr-s from ones with all source operands ready & if available FU exists
  - Selection Policies
  - WinSize ↗ ➔ Search Space ↗
  - # of FUs ↗ ➔ # of Selections↗
- ❖Delay α
  Window Size & # of FUs & Selection Policy

Register File



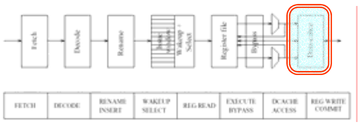
- ❖Previously studied in [6]
- ❖Access Time α
  # of Physical registers & # of read+write ports
- ❖Delay α Issue Width

Data Bypass Logic

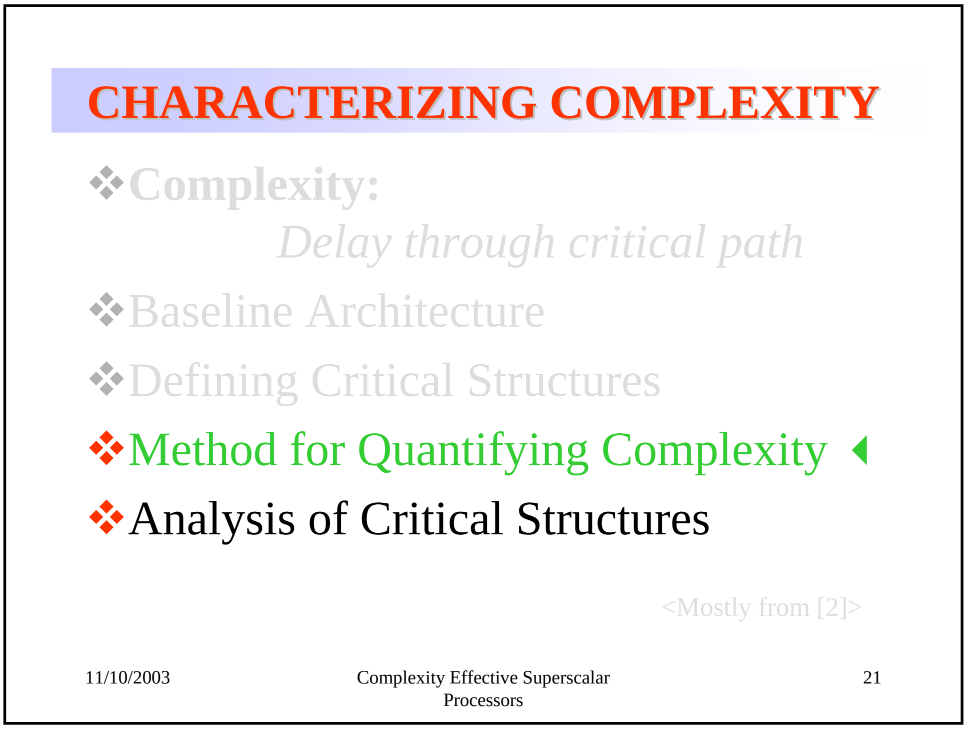


- ***Result Wires:*** Set of wires to bypass results of completed but not committed instr-s
  - # of FUs ↗ → wire lengths↗
  - Pipeline Depth↗ → # of wires↗ & load on wires↗
- ***Operand MUXes:*** select appropriate values to FU I/p ports
  - # of FUs ↗ → Fan-in of MUXes↗
  - Pipeline Depth ↗ → Fan-in of MUXes↗
- Delay α Pipeline depth & # of FUs

Caches



- Studied in [7] & [8]
- [7] gives detailed low level access time analysis
- [8] based on [7]’s methodology, with finer detail
- Delay α Cache Size & Associativity

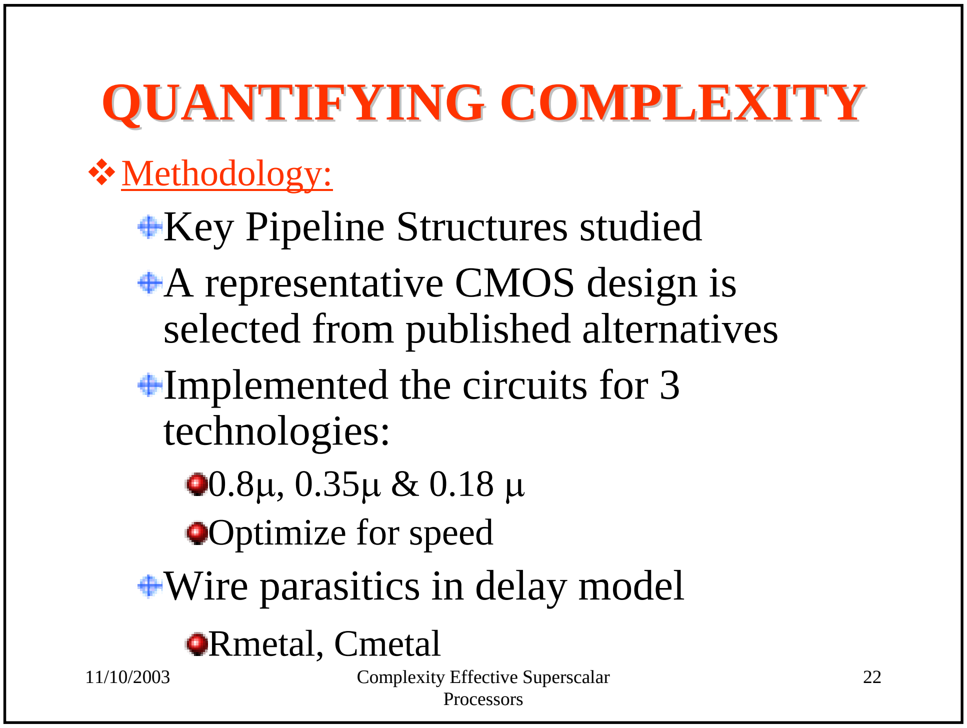


CHARACTERIZING COMPLEXITY

- Complexity:

  Delay through critical path
- Baseline Architecture
- Defining Critical Structures
- Method for Quantifying Complexity ◀
- Analysis of Critical Structures

<Mostly from [2]>



QUANTIFYING COMPLEXITY

- Methodology:
  - Key Pipeline Structures studied
  - A representative CMOS design is selected from published alternatives
  - Implemented the circuits for 3 technologies:
    - 0.8μ, 0.35μ & 0.18 μ
    - Optimize for speed
  - Wire parasitics in delay model
    - Rmetal, Cmetal

QUANTIFYING COMPLEXITY

- Technology Trends:
  - Shrinking Feature Sizes → Scaling
    - Feature size scaling: 1/S
    - Voltage scaling: 1/U
- Logic Delays: $Delay_{gate} = (C_L \times V)/I$
  - C_L: Load Cap.: 1→ 1/S
  - V: Supply Voltage: 1→ 1/U
  - I: Average charge/discharge current: 1→ 1/U
  - Overall Scale factor: 1/S

QUANTIFYING COMPLEXITY

- Wire Delays:
  - L: wire length
  - Intrinsic RC delay →

$$Delay_{wire} = 0.5 \times R_{metal} \times C_{metal} \times L^2$$

    - Rmetal: Resistance per unit length

$$R_{metal} = \rho/(width * thickness)$$

    - Cmetal: Capacitance per unit length

$$\begin{aligned} C_{metal} &= C_{fringe} + C_{parallel-plate} \\ &= 2 * \epsilon * \epsilon_0 * thickness/width + 2 * \epsilon * \epsilon_0 * width/thickness \end{aligned}$$

    - 0.5: 1st order approximation of distributed RC model

QUANTIFYING COMPLEXITY

- Scaling Wire Delays:
  - Metal Thickness doesn’t scale much
  - Width α 1/S
    - Rmetal α S
  - Fringe Capacitance dominates in smaller feature sizes
    - Cmetal α S
  - (Length scales with 1/S)
  - Overall Scale factor: S.S.$(1/S)^2$ = 1

CHARACTERIZING COMPLEXITY

- Complexity:

  Delay through critical path
- Baseline Architecture
- Defining Critical Structures
- Method for Quantifying Complexity ◀
- Analysis of Critical Structures ◀

<Mostly from [2]>

COMPLEXITY ANALYSIS

- Analyzed Structures:
  - Register Rename Logic
  - Wakeup Logic
  - Selection Logic
  - Data Bypass Logic
- Analysis :
  - Logical function
  - Implementation Schemes
  - Delay in terms of μArchitecture Paramaters→
    - Issue Width
    - Window Size

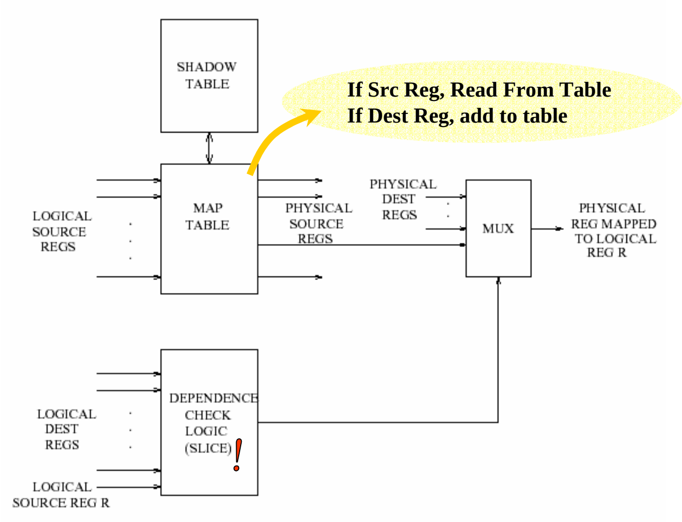
Register Rename Logic

- **Map Table:** Logical Name → Physical Reg.
  - Multiported
    - Multiple instr-s with multiple operands
- **Dependence Check Logic:** Compare each source register to dest. Reg-s of earlier instr-s in current set
  - Multiported
    - Multiple instr-s with multiple operands
- **Shadow Table:** Checkpoint old mappings to recover from branch mispredictions

Register Rename Logic

-

Decoded Instructions



If Src Reg, Read From Table
If Dest Reg, add to table

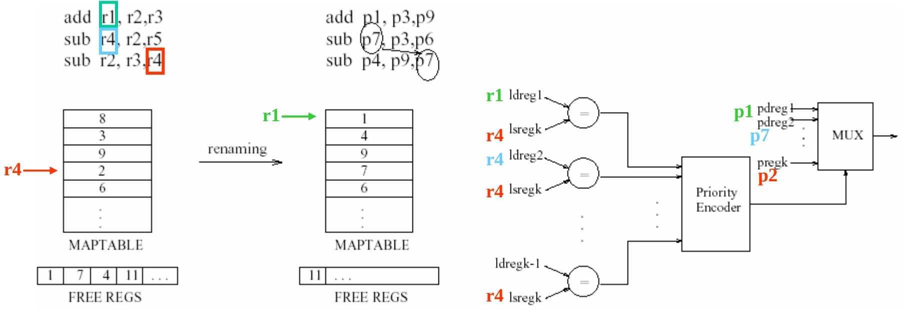
Go to issue window

Map Table Implementation

- Implementation → RAM or CAM
- RAM: (Cross Coupled inverters)
  - Indexed by Logical reg-s = # of entries
  - Entries: Physical reg-s
  - Shift-Register for Checkpointing
- CAM:
  - Associatively searched with logical reg designator
  - Entries: Logical Reg | Valid Bit
  - # of entries = # of physical registers
- CAM vs RAM
  - Similar performance <Only RAM analyzed>

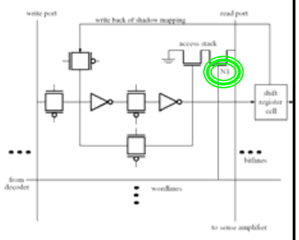
Dependence Check Logic

- ❖ Accessed in Parallel with Map Table
- ❖ Every Logical Reg compared against logical dest regs of current rename group
- ❖ For IW=2,4,8, delay less than map table



Rename Logic Delay Analysis

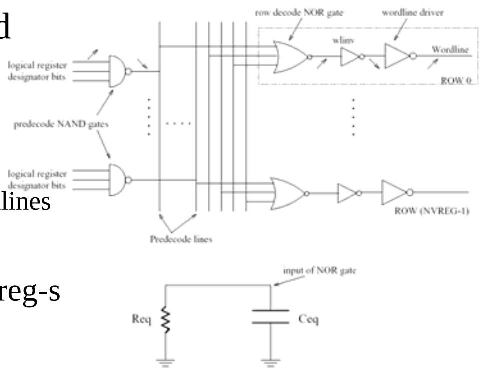
- ❖ Map Table → RAM scheme
- ❖ Delay Components:
  - Time to decode the logical reg index
  - Time to drive wordline
  - Time to pull down bit line
  - Time for SenseAmp to detect pull-down
  - MUX time ignored as control from dep. Check logic comes in advance



$$Delay = T_{decode} + T_{wordline} + T_{bitline} + T_{senseamp}$$

Rename Logic Delay Analysis

- **Decoder Delay:**
- Predecoding for speed
- Length of predecode lines:
  - Cellheight: Height of single cell excluding wordlines
  - Wordline spacing
  - NVREG: # of virtual reg-s
  - x3: 3-operand instr-s



$$PredeclineLength = (cellheight + 3 \times IW \times wordline_{spacing}) \times NVREG$$

Rename Logic Delay Analysis

- **Decoder Delay:**

$$T_{decode} = T_{nand} + T_{nor}$$

- Tnand: Fall delay of NAND

$$T_{nand} = c_0 \times R_{eq} \times C_{eq}$$

- Tnor: rise delay of NOR

$$R_{eq} = R_{nandpd} + 0.5 \times PredeclineLength \times R_{metal}$$

- Rnandpd: NAND pull-down channel resistance
- + Predecode line metal resistance (NAND --- NOR)
  - 0.5 due to distributed R&C model for delay
- Ceq: diff-n Cap. Of NAND + gate Cap. Of NOR + interconnect Cap.➔

$$C_{eq} = C_{diffcap-nand} + C_{gatecap-nor} + PredeclineLength \times C_{metal}$$

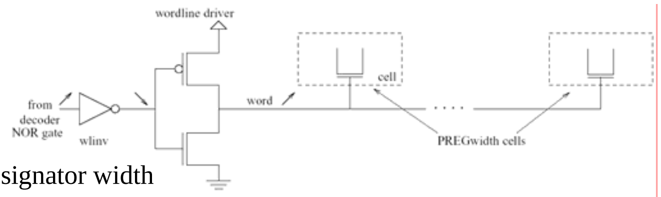
Rename Logic Delay Analysis

- **Decoder Delay:**
- Substituting PredecodeLineLength, Req, Ceq → Tdecode:

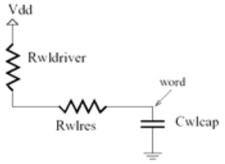
  $\boxed{T_{decode} = c_0 + c \times IW + c_2 \times IW^2}$
- c2: intrinsic RC delay of predecode line
- c2 very small ➔
  - Decoder delay ~linearly dependent on IW

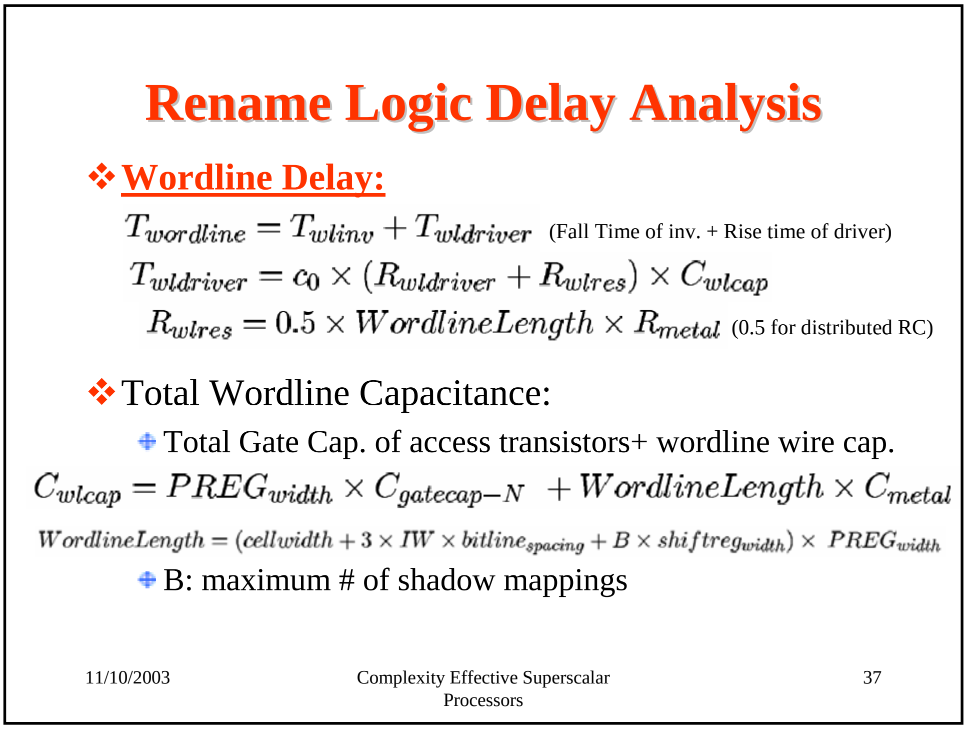
Rename Logic Delay Analysis

- **Wordline Delay:**
- Turn on all access transistors (N1 in cell schematic p.32)



- *PREGwidth*: phys. reg designator width
- *Rwldriver*: pull-up res. Of driver
- *Rwlres*: resistance of wordline
- *Cwlcap*: capacitance on word line





Rename Logic Delay Analysis

- **Wordline Delay:**

$$T_{wordline} = T_{wlinv} + T_{wldriver}$$ (Fall Time of inv. + Rise time of driver)

$$T_{wldriver} = c_0 \times (R_{wldriver} + R_{wlres}) \times C_{wlcap}$$

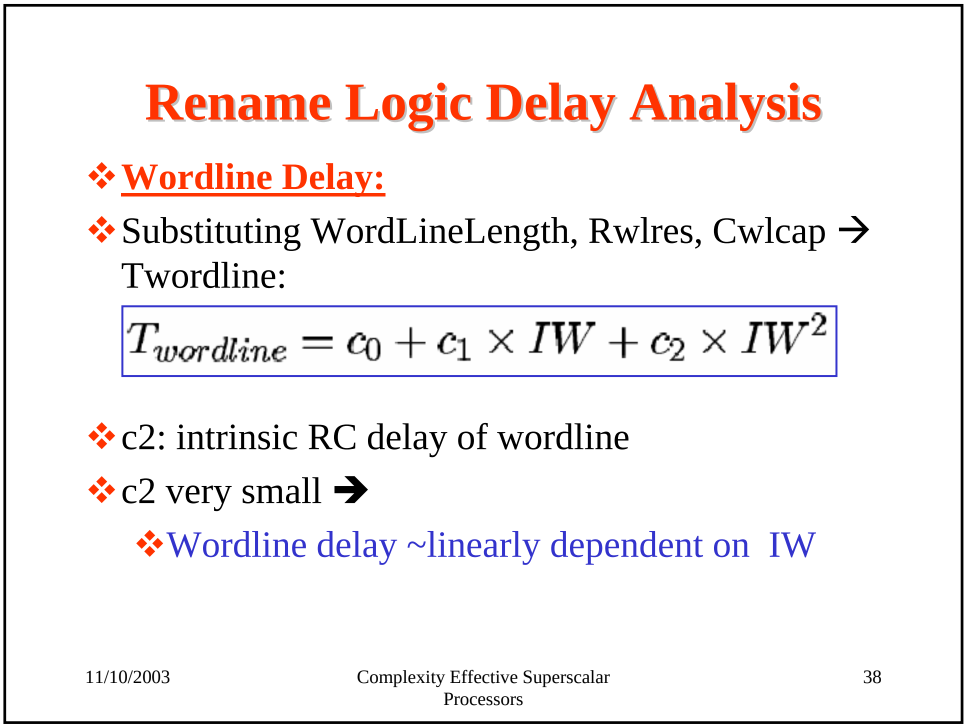
$$R_{wlres} = 0.5 \times WordlineLength \times R_{metal}$$ (0.5 for distributed RC)

- Total Wordline Capacitance:
  - Total Gate Cap. of access transistors+ wordline wire cap.

$$C_{wlcap} = PREG_{width} \times C_{gatecap-N} + WordlineLength \times C_{metal}$$

$$WordlineLength = (cellwidth + 3 \times IW \times bitline_{spacing} + B \times shiftreg_{width}) \times PREG_{width}$$

  - B: maximum # of shadow mappings



Rename Logic Delay Analysis

- **Wordline Delay:**
- Substituting WordLineLength, Rwlres, Cwlcap → Twordline:

$$\boxed{T_{wordline} = c_0 + c_1 \times IW + c_2 \times IW^2}$$

- c2: intrinsic RC delay of wordline
- c2 very small ➔
  - Wordline delay ~linearly dependent on IW

Rename Logic Delay Analysis

- **Bitline Delay:**
- Time from wordline going Hi (Turning on N1) → Bitline going below sense Amp threshold

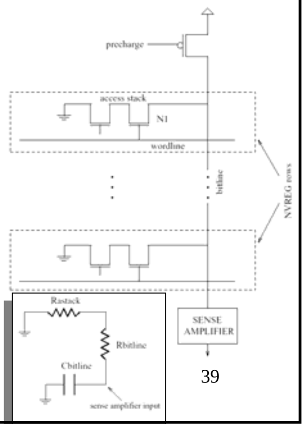
$$T_{bitline} = c_0 \times (R_{astack} + R_{bitline}) \times C_{bitline}$$

$$BitlineLength = (cellheight + 3 \times IW \times wordline_{spacing}) \times NVREG$$

$$C_{bitline} = NVREG \times C_{diffcap-N1} + BitlineLength \times C_{metal}$$

$$R_{bitline} = 0.5 \times BitlineLength \times R_{metal}$$

$$\boxed{T_{bitline} = c_0 + c_1 \times IW + c_2 \times IW^2}$$

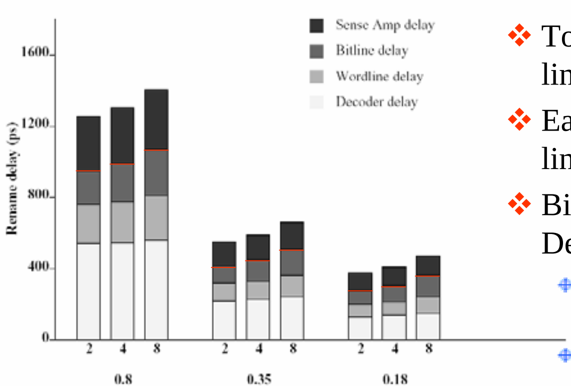


- c2 very small ➔
- Bitline delay ~linearly dependent on IW

Rename Logic Delay Analysis

- **Sense Amplifier Delay:**
- Sense Amp design from [7]
- Implementation ind. of IW
- Delay varies with IW
  - Delay α slope of I/p (bitline Voltage) ➔
  - Delay α bitline delay ➔
- SenseAmp delay ~linearly dependent on IW

Rename Logic Spice Results



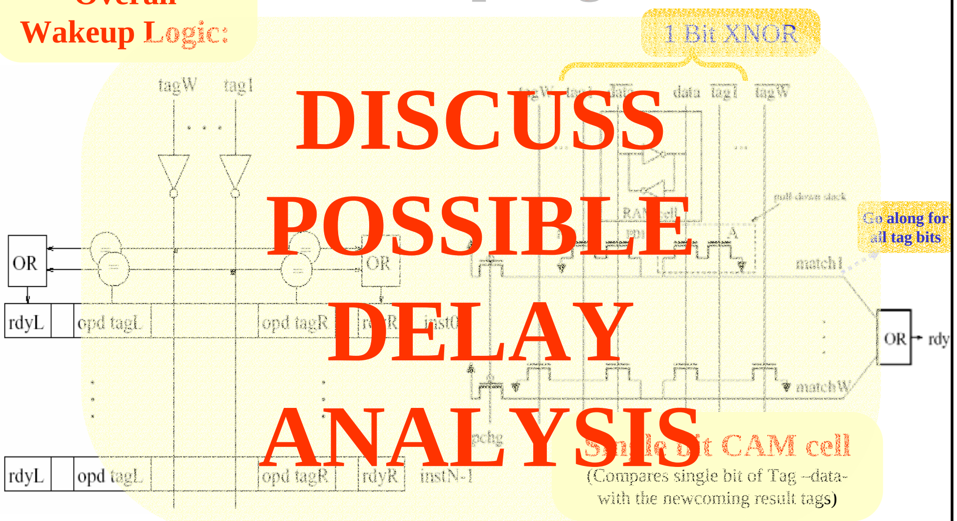
- Total delay increases linearly with IW
- Each Component shows linear increase with IW
- Bitline Delay > Wordline Delay
  - Bitline length α # of Logical reg-s
  - Wordline length α width of physical reg designator
- Feature size ↘ ➔ [increase in bitline&wordline delay with increasing IW] ↗
  - 0.8µ: IW 2→8 ➔ Bitline delay ↗ 37%
  - 0.18µ: IW 2→8 ➔ Bitline delay ↗ 53%

Wakeup Logic

- Updating source dependences for instr-s in issue window
- CAM, 1 instr-n per entry
- When an instr-n produces its result, *tag* associated with the result is broadcast to issue window
  - Each instr-n checks the tag, if matches ➔ sets the corresponding *operand flag*
  - 2 operand/instr-n ➔ 2xIW comparators / entry

Wakeup Logic

Overall Wakeup Logic:



1 Bit XNOR

Go along for all tag bits

DISCUSS POSSIBLE DELAY ANALYSIS

Single bit CAM cell

(Compares single bit of Tag –data- with the newcoming result tags)

Wakeup Logic Delay Analysis

- Critical Path: Mismatch → Pull ready signal low
- Delay Components:
  - Tag drivers → drive tag lines - vertical
  - Mismatched bit: pull down stack → pull matchline low – horizontal
  - Final OR gate → or all the matchlines of an operand tag

$$Delay = T_{tagdrive} + T_{tagmatch} + T_{matchOR}$$

- **$T_{tagdrive}$** α Driver Pullup R & Tagline length & Tagline Load C

$$\boxed{T_{tagdrive} = c_0 + (c\ + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW + c\ \times IW^2) \times WINSIZE^2}$$

  - Intermediate equations here
  - Quadratic component significant for IW>2 & 0.18µ

Wakeup Logic Delay Analysis

- $T_{tagmatch}$ α Pulldown Stack Pulldown R & Matchline length & Matchline Load C

$$\boxed{T_{tagmatch} = c_0 + c_1 \times IW + c_2 \times IW^2}$$

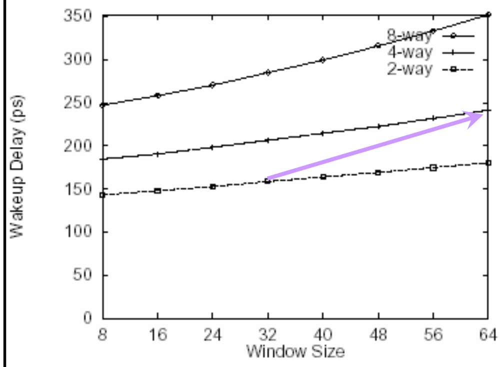
  - Intermediate equations here

- $T_{matchOR}$ α Fan-in (Delay of a gate α Fan-in^2)
  - <Worst Case Fan-in^2 RC>

$$\boxed{T_{matchOR} = c_0 + c_1 \times IW + c_2 \times IW^2}$$

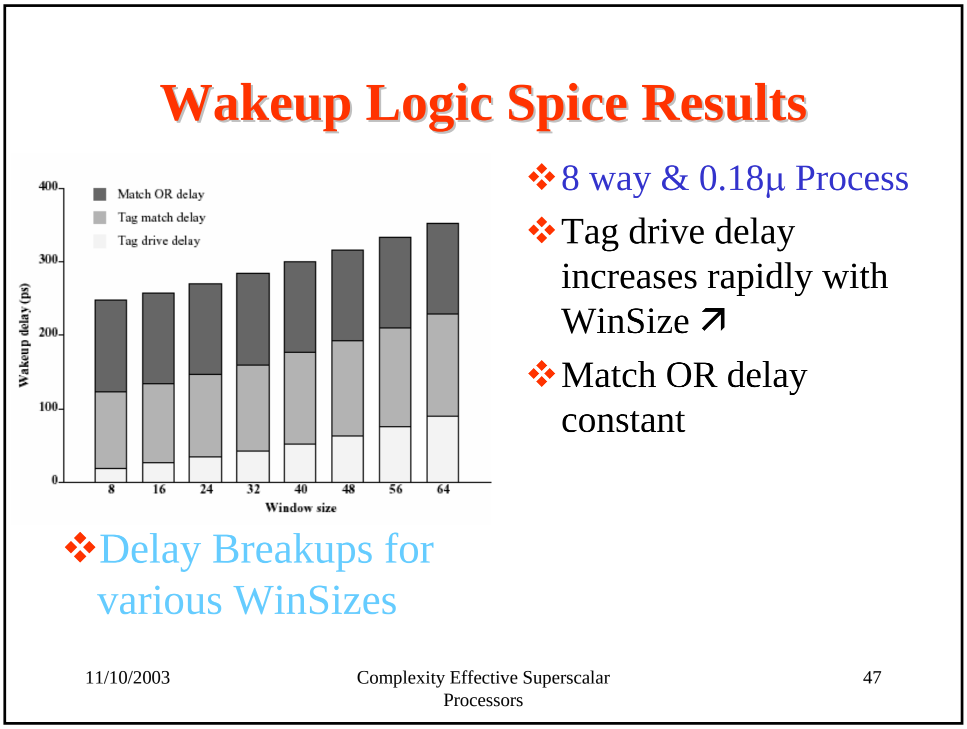
  - Quadratic component Small for both cases
  - Both delays ~linearly dependent on IW

Wakeup Logic Spice Results



- Delay wrt Window size & Issue width

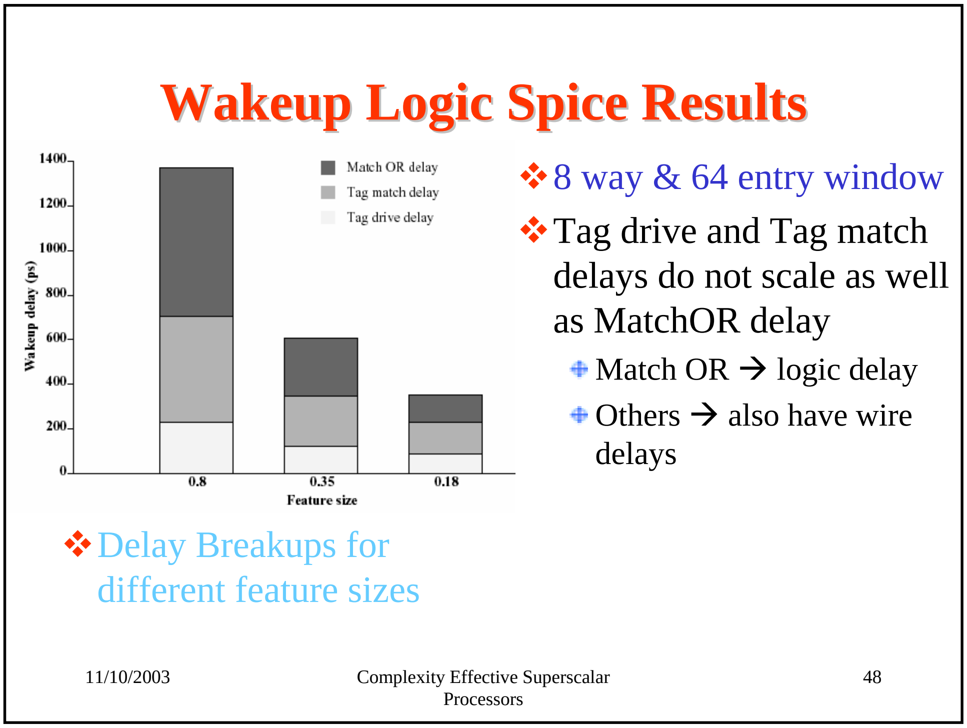
- 0.18μ Process
- Quadratic dependence
- Issue width has greater effect → increase all 3 delay components
- As IW & WinSize ↗ together → delay actually changes like: THIS



Wakeup Logic Spice Results

- 8 way & 0.18µ Process
- Tag drive delay increases rapidly with WinSize ↗
- Match OR delay constant

- Delay Breakups for various WinSizes



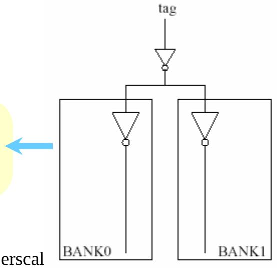
Wakeup Logic Spice Results

- 8 way & 64 entry window
- Tag drive and Tag match delays do not scale as well as MatchOR delay
  - Match OR → logic delay
  - Others → also have wire delays

- Delay Breakups for different feature sizes

Wakeup Logic Spice Results

- All simulations have max WinSize 64
  - Larger Window ➔ Tagline RC delay ↗ ↗ (Tagline RC delay α WinSize^2)
- For larger windows ➔ Use **Window Banking**
  - Reduces Tagline length

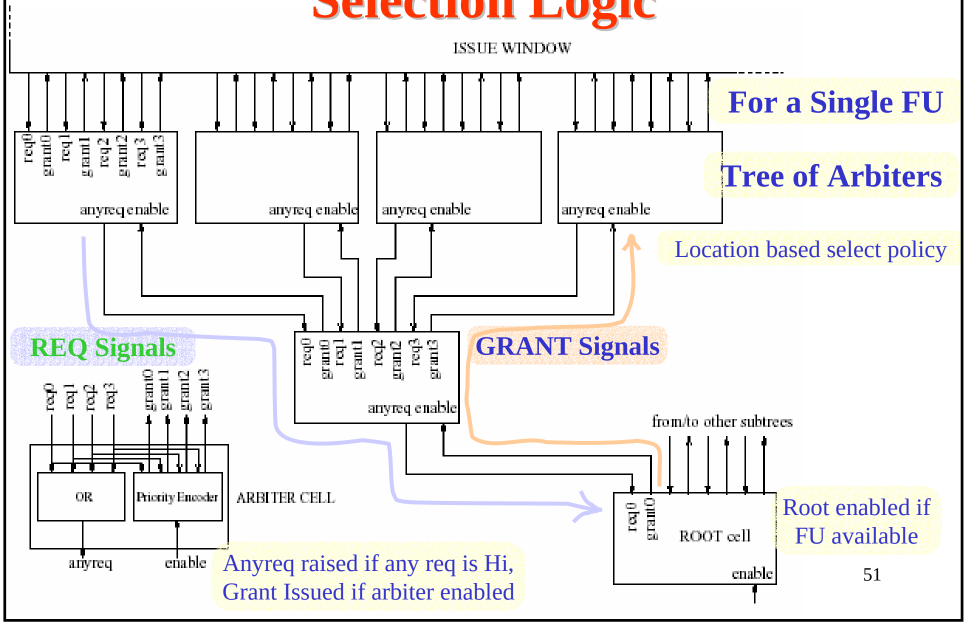


Improves RC Delay by ~x(1/4)

Selection Logic

- Chooses ready instructions to issue
  - Might be up to WinSize ready instr-s
  - Instr-s need to be steered to specific FUs
- I/p → **REQ**:
  - Produced by wakeup logic when all operands ready
  - 1 per instr-n in issue window
- O/p → **GRANT**:
  - Grants issue to requesting instr-n
  - 1 per request
- Selection Policy

Selection Logic



For a Single FU

Tree of Arbiters

Location based select policy

REQ Signals

GRANT Signals

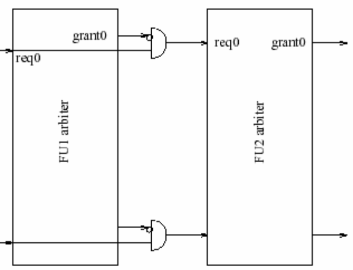
Root enabled if FU available

Anyreq raised if any req is Hi, Grant Issued if arbiter enabled

Selection Logic

❖Handling Multiple FUs of Same Type:

- Stack Select logic blocks in series - hierarchy
- Mask the Request granted to previous unit



- NOT Feasible for More than 2 FUs
  - Alternative: statically partition issue window among FUs – MIPS R10000, HP PA 8000

Selection Logic Delay Analysis

- Delay: time to generate GRANT after REQ
- Delay Components:
  - Time for REQ to propagate: instr-n → Root
  - Root Delay
  - Time for GRANT to propagate: Root → instr-n

$$Delay = (L-1) \times T_{reqpropd} + T_{root} + (L-1) \times T_{grantpropd}$$

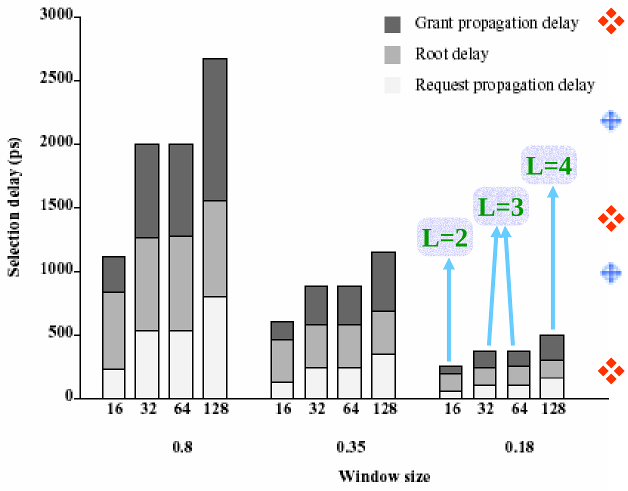
- (L: Depth of Arrbiter Tree)

- 4 I/p arbiter cells Optimum → $L = \log_4(WINSIZE)$

→ $\boxed{Delay = c_0 + c_1 \times log_4(WINSIZE)}$

- Delay ~logarithmically dependent on WinSize

Selection Logic Spice Results



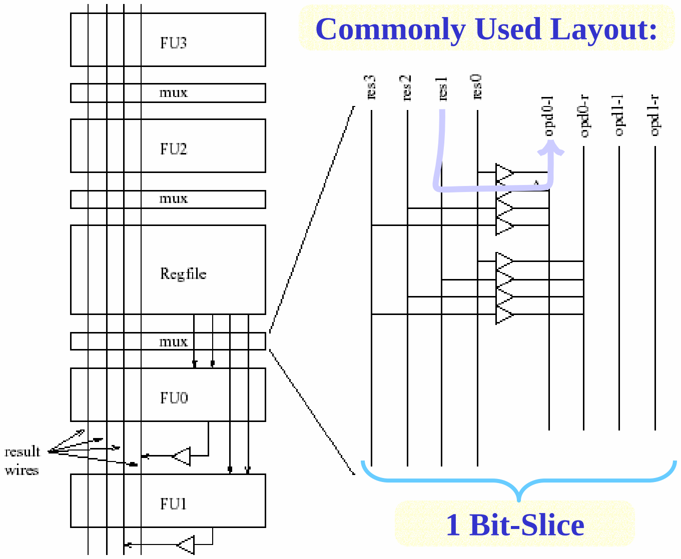
- Root delay same for each WinSize →
  - L↗ x2 → Delay↗ < x2
- Logic Delays →
  - Scale well with feature size
- Caution!: Wire delays not included!

Data Bypass Logic

- Result Forwarding
- Number of possible bypasses:
  - S pipestages after first result stage & 2 I/p FUs

    ➔ $2 \times IW^2 \times S$
- Key Delay Component:
  - Delay of result wires → bypass length & load
  - Strongly layout dependent

Data Bypass Logic



Commonly Used Layout:

Turn on Tri-State A to pass result of FU1 to left operand of FU0

1 Bit-Slice

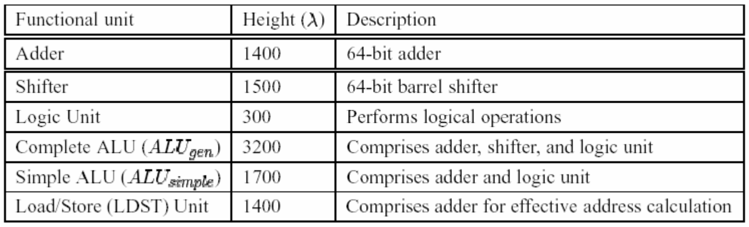
Data Bypass Logic Delay Analysis

- Delay → Generic wire delay:

$$Delay_{wire} = 0.5 \times R_{metal} \times C_{metal} \times L^2$$

  - L is dependent on # of FUs (IW) & FU heights
  - Pipeline depth↗ → C ↗ **<NOT implemented in simulations!>**

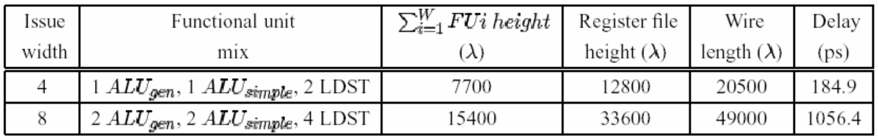
- Typical FU heights:



Functional unit	Height (λ)	Description
Adder	1400	64-bit adder
Shifter	1500	64-bit barrel shifter
Logic Unit	300	Performs logical operations
Complete ALU (ALU_{gen})	3200	Comprises adder, shifter, and logic unit
Simple ALU (ALU_{simple})	1700	Comprises adder and logic unit
Load/Store (LDST) Unit	1400	Comprises adder for effective address calculation

Data Bypass Logic Delay Analysis

- Computed delays for hypothetical machines:



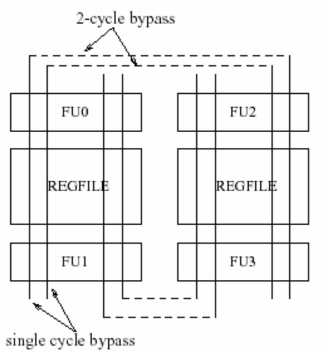
Issue width	Functional unit mix	$\sum_{i=1}^{W} FUi\ height$ (λ)	Register file height (λ)	Wire length (λ)	Delay (ps)
4	1 ALU_{gen}, 1 ALU_{simple}, 2 LDST	7700	12800	20500	184.9
8	2 ALU_{gen}, 2 ALU_{simple}, 4 LDST	15400	33600	49000	1056.4

  - (Delay independent of feature size)

- Delay dependent on (IW)^2

Data Bypass Logic Alternative Layouts

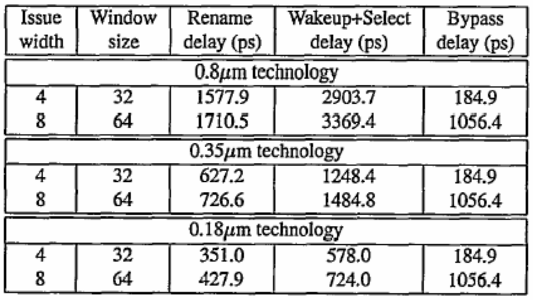
- ❖Delay computation directly dependent on layout
  - Future → Clustered Organizations (DEC 21264)



- Each cluster of FUs with its own regfile
- Intra-Cluster bypasses: 1 cycle
- Inter-Cluster bypasses: 2 or more cycles
  - µArch & compiler effort to ensure inter cluster bypasses occur infrequently

CHARACTERIZING COMPLEXITY

❖**Summary:**



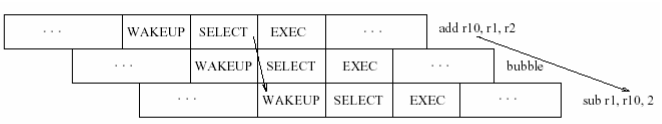
Issue width	Window size	Rename delay (ps)	Wakeup+Select delay (ps)	Bypass delay (ps)
0.8μm technology				
4	32	1577.9	2903.7	184.9
8	64	1710.5	3369.4	1056.4
0.35μm technology				
4	32	627.2	1248.4	184.9
8	64	726.6	1484.8	1056.4
0.18μm technology				
4	32	351.0	578.0	184.9
8	64	427.9	724.0	1056.4

- 4 Way → Window Logic is bottleneck
- 8 Way → Bypass Logic is bottleneck

CHARACTERIZING COMPLEXITY

- **Summary:**
  - Future → Window logic**!** & Bypass logic**!**
  - Both are ‘atomic’ operations:

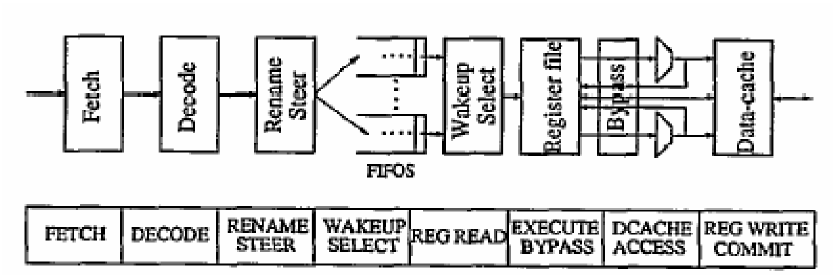
    - dependent instr-s cannot issue consecutively if pipelined



COMPLEXITY EFFECTIVE MICROARCHITECTURE

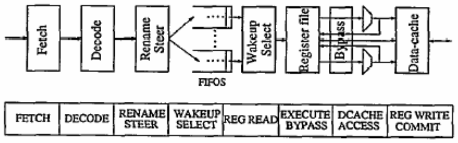
- Brainiac & Maniac
  - High IPC & High CLK rate
- Simplify Wakeup & Selection Logics
- Naturally extendable to clustering →
  - Can solve bypass problem
- Group dependent instr-s rather than independent ones →
- Dependence Based Architecture

DEPENDENCE ARCHITECTURE



- Dependent instr-s cannot execute in parallel
- Issue Window → FIFO buffers (issue inorder)
  - '*Steer*' dependent instr-s to same FIFO
- Only FIFO heads need check for ready operands

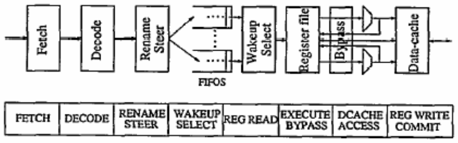
DEPENDENCE ARCHITECTURE



- **SRC_FIFO Table:**
  - Similar to Map table
  - Indexed with logical register designator
  - Entries: SRC-FIFO(Rs)=FIFO where the instr-n that will write Rs exists. <Invalid if instr-n completed>
  - Can be accessed parallel with map table

DEPENDENCE ARCHITECTURE

❖ **Steering Heuristic:**

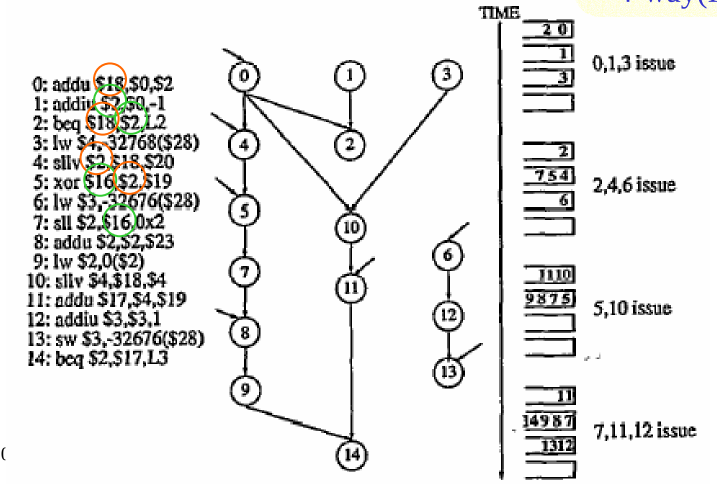


- If all operands of instr-n in regfile➔ Steer to an empty FIFO
- Instr-n has a single outstanding operand to be written by Inst0, in FIFO F0 ➔
  - No instr-n behind Inst0 ➔ steer to Fa
  - O/w ➔ steer to an empty FIFO
- Instr-n has 2 outstanding operands to be written by Inst0&Inst1 in Fa & Fb ➔
  - No instr-n behind Inst0 ➔ steer to Fa
  - O/w → No instr-n behind Inst1 ➔ steer to Fb
  - O/w ➔ steer to an empty FIFO
- If all FIFOs full/No Empty FIFOs ➔ STALL

DEPENDENCE ARCHITECTURE

❖ **Steering Heuristic <Ex>:**

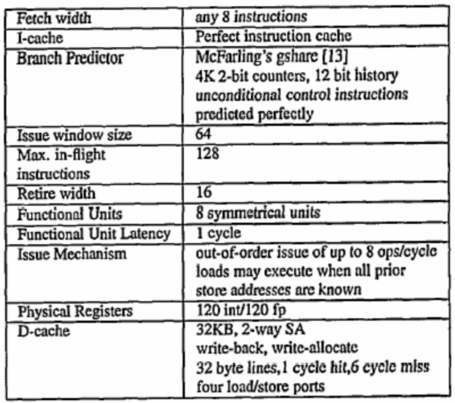
Steer Width: 4
4-way(IW)



```
 0: addu $18,$0,$2
 1: addiu $2,$0,-1
 2: beq $18,$2,L2
 3: lw $4,-32768($28)
 4: sllv $2,$18,$20
 5: xor $16,$2,$19
 6: lw $3,-32676($28)
 7: sll $2,$16,0x2
 8: addu $2,$2,$23
 9: lw $2,0($2)
10: sllv $4,$18,$4
11: addu $17,$4,$19
12: addiu $3,$3,1
13: sw $3,-32676($28)
14: beq $2,$17,L3
```

Performance Results

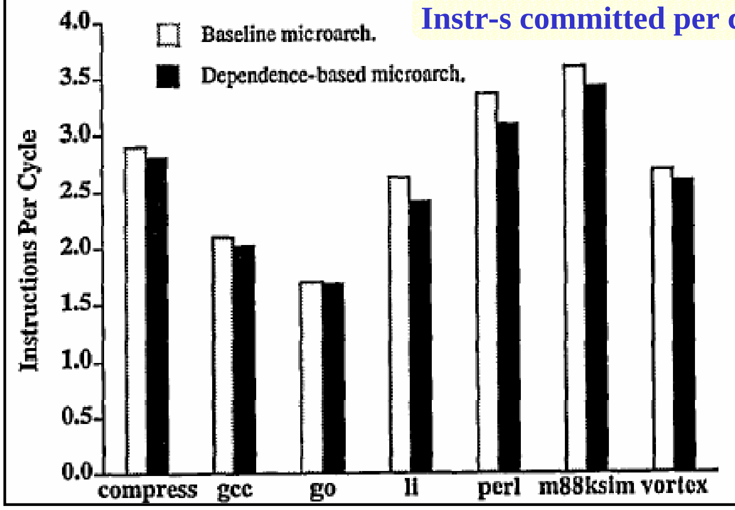
- **Dependence Arch. vs. Baseline**
  - 8 FIFOs, 8 entries/ FIFO vs. WinSize=64
  - 8 –way, aggressive instr-n fetch (no block)
  - SimpleScalar Simulation →
  - SPEC’95
  - 0.5B instr-s



Fetch width	*any 8 instructions*
I-cache	Perfect instruction cache
Branch Predictor	McFarling’s gshare [13] 4K 2-bit counters, 12 bit history unconditional *control instructions* predicted perfectly
Issue window size	64
Max. in-flight instructions	128
Retire width	16
Functional Units	8 symmetrical units
Functional Unit Latency	1 cycle
Issue Mechanism	out-of-order issue of up to 8 ops/cycle loads may execute when all prior store addresses are known
Physical Registers	120 int/120 fp
D-cache	32KB, 2-way SA write-back, write-allocate 32 byte lines,1 cycle hit,6 cycle miss four load/store ports

Performance Results

- **Dependence Arch. vs. Baseline:**



Instr-s committed per cycle

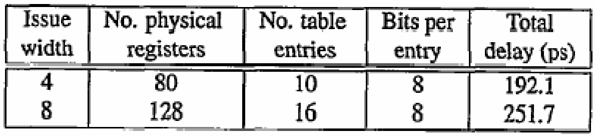
Max Performance Degradation 8% in li

Complexity Analysis

- **Wakeup Logic:**
  - Need not to broadcast result tags to all window entries → only to FIFO heads
  - Reservation Table:
    - 1 bit per reg→ ‘Waiting for data’
      - Set result reg when instr-n dispatched
      - Clear when instr-n executes
    - Instr-n at FIFO head checks its operands’ bits
  - Delay of Wakeup logic →
    Delay of Reservation table access

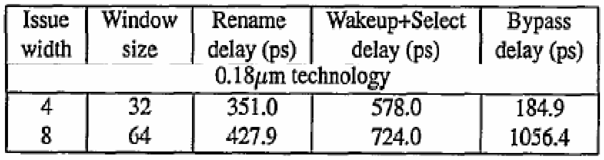
Complexity Analysis

- Reservation Station vs. Baseline Wakeup:
- Reservation Station: 80 Regs, 0.18μ:



Issue width	No. physical registers	No. table entries	Bits per entry	Total delay (ps)
4	80	10	8	192.1
8	128	16	8	251.7

- Window-Based arch. 32&64 Regs:



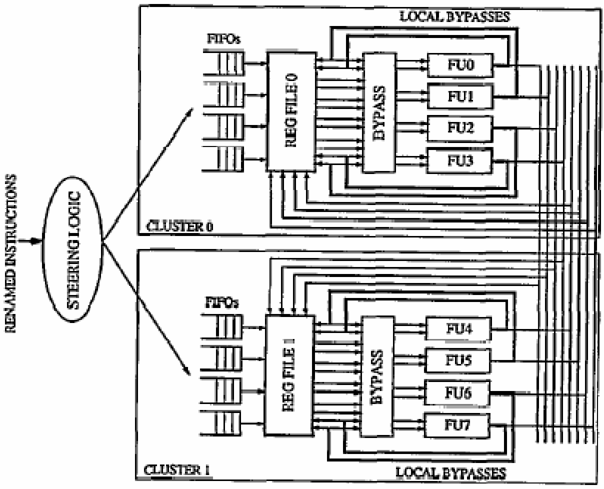
Issue width	Window size	Rename delay (ps)	Wakeup+Select delay (ps)	Bypass delay (ps)
0.18μm technology				
4	32	351.0	578.0	184.9
8	64	427.9	724.0	1056.4

Complexity Analysis

- **Instruction Steering:**
- Done parallel with renaming
- SRC-FIFO table smaller than rename table
  - Smaller delay
- **Summary:**
  - Wakeup-Select Delay reduced
  - Faster clock rate ~39%
  - IPC Performance degrade < 8%
    - → ~ **27% execution speed advantage**

Clustered Architecture

- 2x4 way:



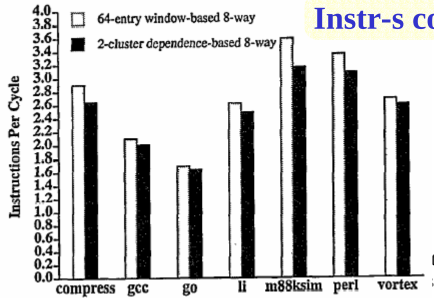
- Local Bypass → single cycle
- Inter cluster bypass → > 1 cycle
- Regfiles identical, within a cycle delay

Clustered Architecture

- **Advantages:**
  - Wakeup-Select Function already simplified
  - Steer Heuristic → Dependent instr-s to same FIFO ➔ less inter cluster bypasses
  - Critical bypass logic delay reduced – Main motivation of clustering
  - Regfile Access delay reduced as # of ports ↘
- Heuristic Modified:
  - Two separate free FIFO lists for each cluster

Clustered Architecture Performance

- 2x4 way Dependence Arch. vs. 8-way baseline architecture
  - 2x4 8-entry FIFOs vs. 64 entry window
  - Inter-cluster bypass → 2 cycles vs. all single cycle bypasses



Instr-s committed per cycle

Max Performance Degradation 12% in m88ksim

Clustered Architecture Performance

- ❖Dependence Arch will have higher clock rate: > 4-way, WinSize 32, baseline ➔

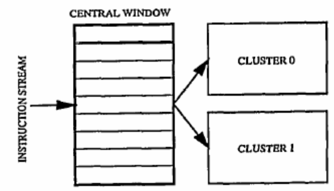
$$\frac{Speed_{dependenceArch}}{Speed_{WindowArch}} = \frac{Delay\ \ of\ \ 8\ \ way\ \ 64\ \ entry\ \ window}{Delay\ \ of\ \ 4\ \ way\ \ 32\ \ entry\ \ window} = \frac{724}{578} \cong 1.25$$

- ❖Potential Speedup over Window based architecture > 88% x 125% = 110%

- ❖**More than 10% performance improvement over baseline**

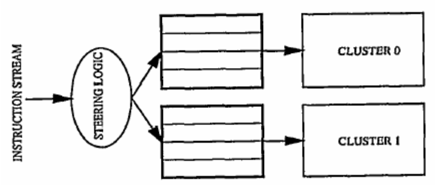
Other Clustered Architectures

- ❖In all cases, inter cluster bypass → 2 cycles
- ❖**1) Single Window, Execution Driven Steering:**
  - Steer to cluster which provides the source operands first
  - Higher IPC than double window
  - Back to the complex wakeup-select logic ☹



Other Clustered Architectures

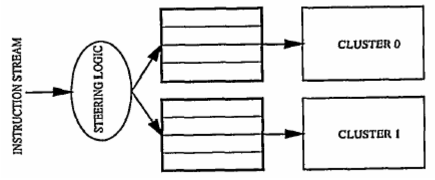
❖2) 2 Windows, Dispatch Driven Steering:



- Similar to dependence architecture
- Random access windows rather than FIFOs
- Steer with a similar dependence heuristic
- Still somewhat complex wakeup-select logic ☹

Other Clustered Architectures

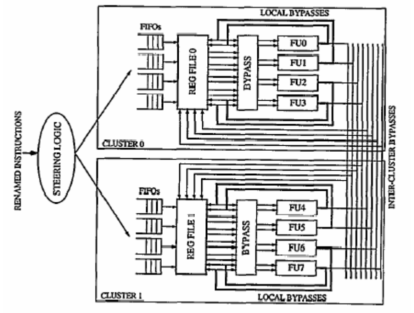
❖3) 2 Windows, Random Steering:



- Same as dispatch driven architecture
- Steer randomly
- For Theoretical baseline comparison

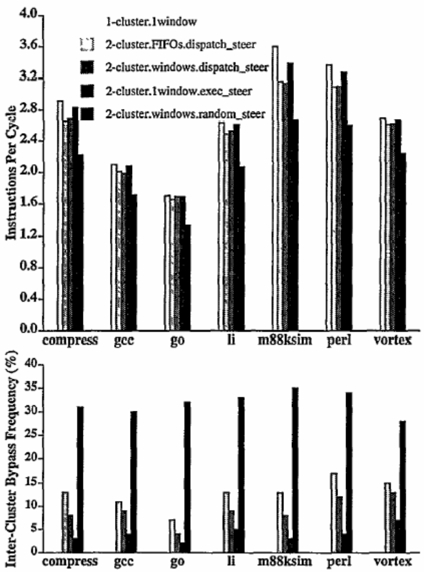
Other Clustered Architectures

❖ 4) Clustered Dependence Architecture→
2 Set of FIFOs, Dispatch Driven Steering:



- Simple Wakeup Select Logic ☺

Performance Comparison



❖ Ideal → 64 entry window, single bypass all

❖ Others → WinSize:1) 64x1 2)32x2 3)32x2 4)(4x8)x2

❖ Max performance degradation 26% (m88ksim)

❖ Almost always as well as 2 windows dispatch driven steer

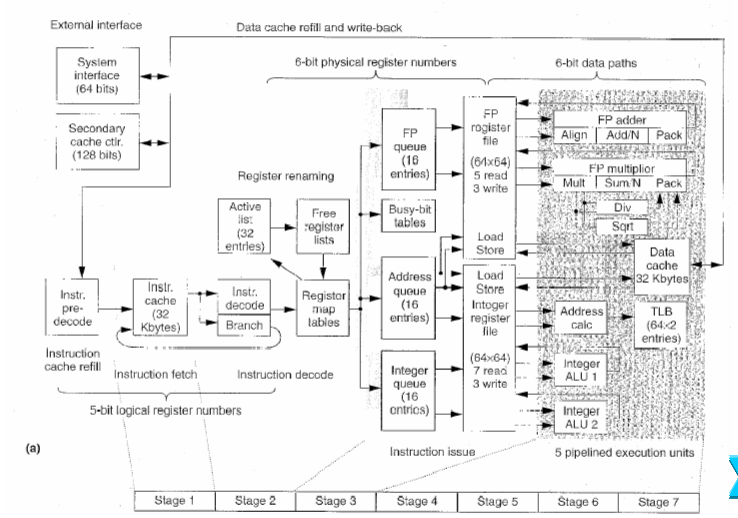
❖ Suspicion: m88ksim FIFO does better than 2 window dispatch driven steer?

Conclusions

- ❖ Window & bypass logic are future (for 1997) performance bottlenecks
- ❖ Clustered Dependence Based Architecture Performs with little IPC degradation, additional clock speed aggregates 16% speedup over current baseline model.
- ❖ Wider IW and smaller feature sizes will empasize this speedup

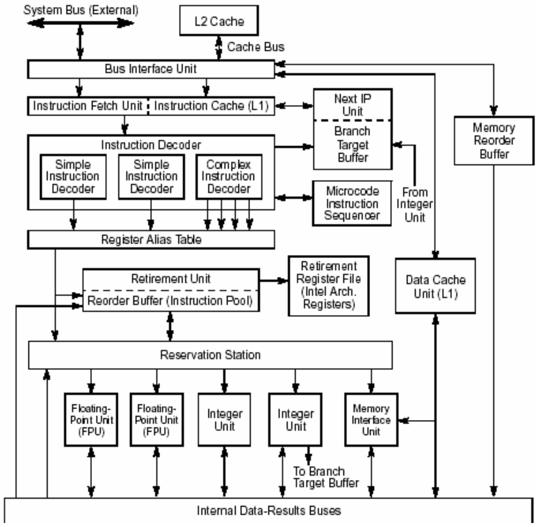
ADDITIONAL SLIDES

MIPS R10000 PIPELINE



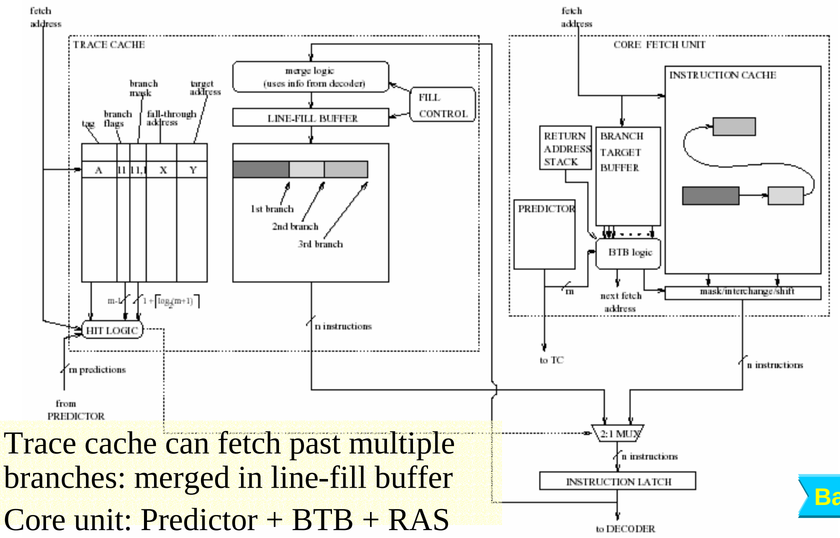
Back

INTEL P6 PIPELINE



Back

INSTRUCTION FETCH LOGIC



- ❖ Trace cache can fetch past multiple branches: merged in line-fill buffer
- ❖ Core unit: Predictor + BTB + RAS

Back

Register File Complexity Analysis [6]

- ❖ Analysis for 4 way & 8 way processors
  - 4 way → 32 Entry Issue Window
  - 8 way → 64 Entry Issue Window
- ❖ Different Register File Organizations
  - Issue Width → # of Read/Write Ports
    - 4 way → Integer Regfile:
      8 Read & 4 Write Ports
      Floating Point Regfile:
      4 Read & 2 Write Ports
    - 8 way → Integer Regfile:
      16 Read & 8 Write Ports
      Floating Point Regfile:
      8 Read & 4 Write Ports
  - Different Regfile sizes

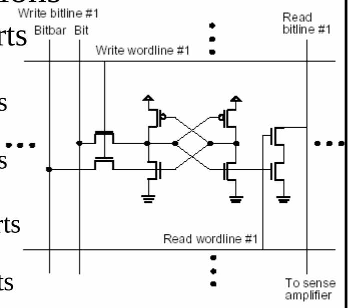
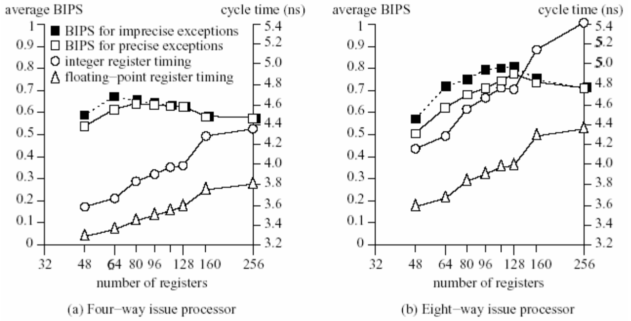


Figure 9: Multiported register file cell.

Back

Register File Complexity Analysis [6]



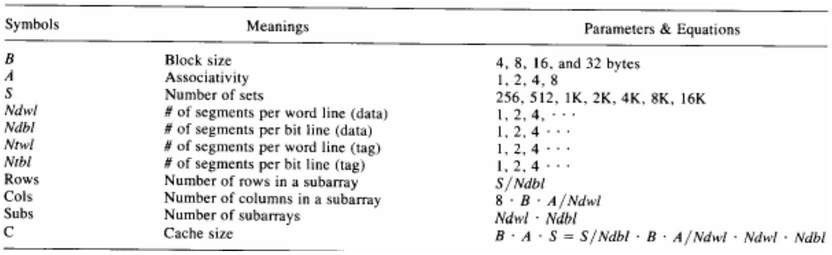
(a) Four–way issue processor

(b) Eight–way issue processor

- ❖ FP Regfile faster than Int Regfile ← Less Ports
- ❖ Doubling number of ports ➔
  Double # of wordlines and bitlines
  - Quadruple Regfile Area
- ❖ Doubling number of Registers ➔
  Double # of wordlines
  - Double Regfile Area

Back

Cache Access Time [7]

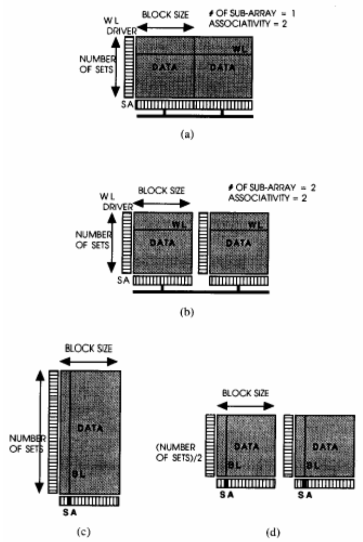


Symbols	Meanings	Parameters & Equations
B	Block size	4, 8, 16, and 32 bytes
A	Associativity	1, 2, 4, 8
S	Number of sets	256, 512, 1K, 2K, 4K, 8K, 16K
Ndwl	# of segments per word line (data)	1, 2, 4, · · ·
Ndbl	# of segments per bit line (data)	1, 2, 4 · · ·
Ntwl	# of segments per word line (tag)	1, 2, 4 · · ·
Ntbl	# of segments per bit line (tag)	1, 2, 4 · · ·
Rows	Number of rows in a subarray	$S/Ndbl$
Cols	Number of columns in a subarray	$8 \cdot B \cdot A/Ndwl$
Subs	Number of subarrays	$Ndwl \cdot Ndbl$
C	Cache size	$B \cdot A \cdot S = S/Ndbl \cdot B \cdot A/Ndwl \cdot Ndwl \cdot Ndbl$

- ❖ Ndwl, Ndbl, Ntwl, Ntbl → Layout parameters
- ❖ Access Time = Decoder Delay + Word-line delay + Bit-line/Sense Amplifier Delay + Data Bus Delay
- ❖ Formula & Derivations in paper
- ❖ Time breakdown plots not descriptive of cache parameters
  - I.e Twl vs. (B.8).A/Ndwl

Back

Cache Access Time [7]

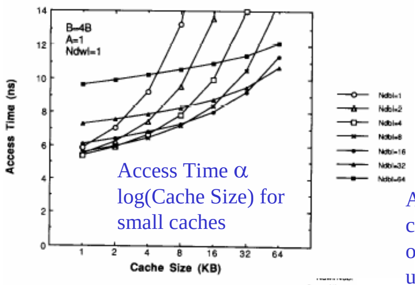


- ❖ Ndwl, Ndbl, Ntwl, Ntbl Layout parameters:

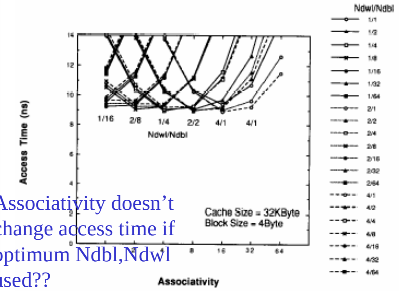
a. 2-Way Set Assoc. (A=2), Ndwl=Ndbl=1
b. A=2, Ndwl=2, Ndbl=1
c. A=1, Ndwl=Ndbl=1
d. A=1, Ndwl=1, Ndbl=2

Back

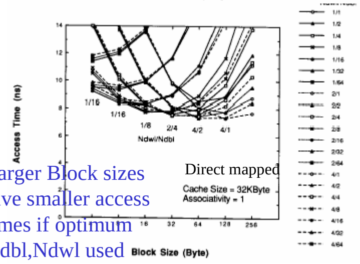
Cache Access Time [7]



Access Time α log(Cache Size) for small caches



Associativity doesn't change access time if optimum Ndbl,Ndwl used??



Direct mapped

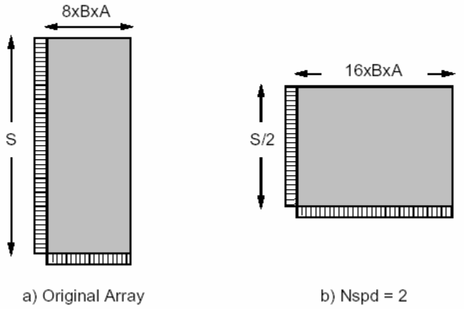
Larger Block sizes give smaller access times if optimum Ndbl,Ndwl used

- ❖ With correct layout parameters: Delay α Access Time, 1/(Block Size), and NOT Associativity

Back

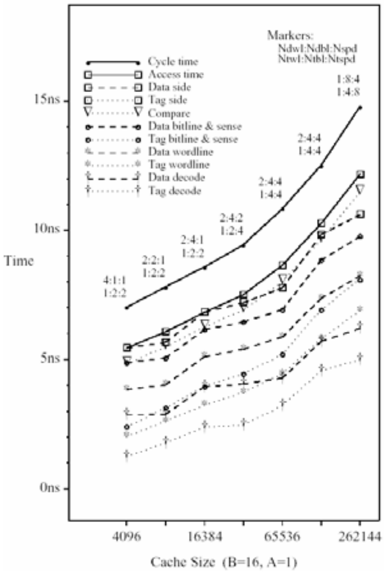
Cache Access Time [8]

- Additional Layout parameters: Nspd & Ntsbd
  - How many sets are mapped to a single wordline



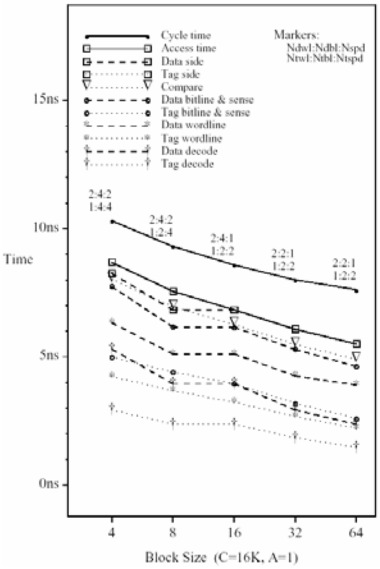
- optimum *Ndwl*, *Ndbl*, and *Nspd* depend on cache and block sizes, and associativity.

Cache Access Time [8]



- Cache Size vs. Access Time:
  - Block size=16 Bytes
  - Direct Mapped Cache
  - For each size, optimum layout parameters used
  - Access time breakdowns are shown
  - Comparator delay significant
  - Cache Size ↗ ➔ Access Time↗

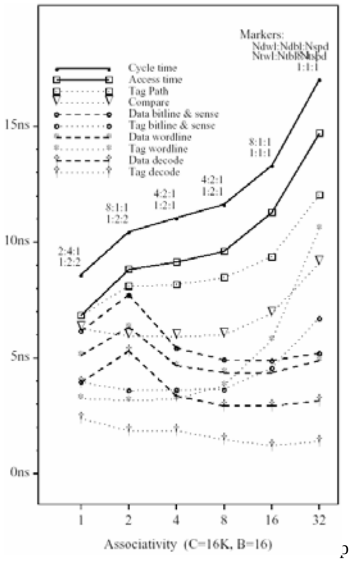
Cache Access Time [8]



- ❖ <u>Block Size vs. Access Time:</u>
  - Cache size=16 KBytes
  - Direct Mapped Cache
  - For each block size, optimum layout parameters used
  - Access time breakdowns are shown
  - Access time ↘ due to drop in decoder delay
  - Block Size ↗ ➔ Access Time ↘

Back

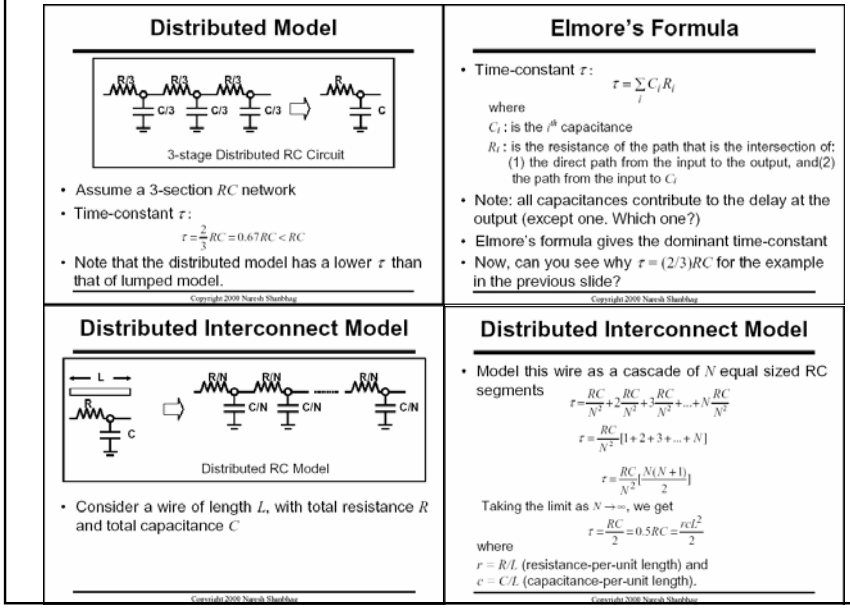
Cache Access Time [8]



- ❖ <u>Associativity vs. Access Time:</u>
  - Cache size=16 KBytes
  - Block Size 16 bytes
  - For each case, optimum layout parameters used
  - Access time breakdowns are shown
  - Associativity ↗ ➔ Access Time ↗

Back

Distributed RC Model



R/3
R/3
R/3
C/3
C/3
C/3
R
C
L
R
C
R/N
R/N
R/N
C/N
C/N
C/N

Distributed Model

3-stage Distributed RC Circuit

- Assume a 3-section RC network
- Time-constant τ:

$$\tau = \frac{2}{3}RC = 0.67RC < RC$$

- Note that the distributed model has a lower τ than that of lumped model.

Elmore's Formula

- Time-constant τ:

$$\tau = \sum_i C_i R_i$$

where

C_i : is the i^{th} capacitance

R_i : is the resistance of the path that is the intersection of: (1) the direct path from the input to the output, and(2) the path from the input to C_i

- Note: all capacitances contribute to the delay at the output (except one. Which one?)
- Elmore's formula gives the dominant time-constant
- Now, can you see why $\tau = (2/3)RC$ for the example in the previous slide?

Distributed Interconnect Model

Distributed RC Model

- Consider a wire of length L, with total resistance R and total capacitance C

Distributed Interconnect Model

- Model this wire as a cascade of N equal sized RC segments

$$\tau = \frac{RC}{N^2} + 2\frac{RC}{N^2} + 3\frac{RC}{N^2} + \ldots + N\frac{RC}{N^2}$$

$$\tau = \frac{RC}{N^2}[1+2+3+\ldots+N]$$

$$\tau = \frac{RC}{N^2}[\frac{N(N+1)}{2}]$$

Taking the limit as $N \to \infty$, we get

$$\tau = \frac{RC}{2} = 0.5RC = \frac{rcL^2}{2}$$

where

$r = R/L$ (resistance-per-unit length) and

$c = C/L$ (capacitance-per-unit length).

Back

Sense Amplifier [7]

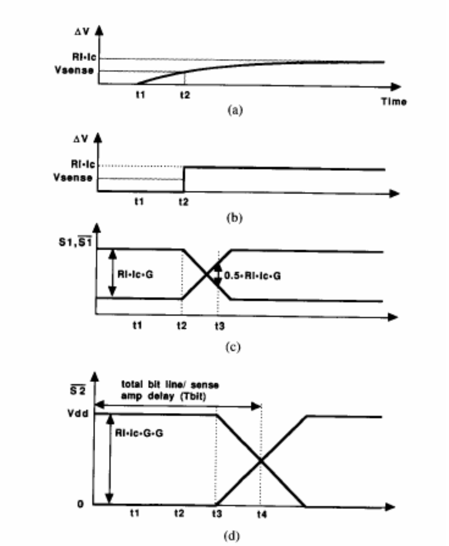


Fig. 9. Sense amplifier. (a) Bit-line voltage difference (ΔV). (b) Approximation of ΔV. (c) Waveforms of S_1 and $\overline{S_1}$. (d) Waveforms of $\overline{S_2}$.

Back

Wakeup Logic Tagline Equations

$$Delay = T_{tagdrive} + T_{tagmatch} + T_{matchOR}$$

$$TaglineLength = (camheight + IW \times matchline\_spacing) \times WINSIZE$$

$$T_{tagdrive} = c_0 \times (R_{tagdriver-pup} + R_{tlres}) \times C_{tlcap}$$

$$R_{tlres} = 0.5 \times TaglineLength \times R_{metal}$$

$$C_{tlcap} = TaglineLength \times C_{metal} + C_{gatecap-comp} \times WINSIZE + C_{diffcap-tagdriver}$$

$$T_{tagdrive} = c_0 + (c \;\; + c_2 \times IW) \times WINSIZE + (c_3 + c_4 \times IW + c \;\; \times IW^2) \times WINSIZE^2$$

Back

Wakeup Logic Matchline Equations

$$T_{tagmatch} = c_0 \times (R_{pdstack} + R_{mlres}) \times C_{mlcap}$$

$$R_{mlres} = 0.5 \times MatchlineLength \times R_{metal}$$

$$MatchlineLength = (camwidth + IW \times tagline_{spacing}) \times PREG_{width}$$

$$C_{mlcap} = 2 \times PREG_{width} \times C_{diffcap-PD1} + MatchlineLength \times C_{metal} + C_{gatecap\_matchinv}$$

$$T_{tagmatch} = c_0 + c_1 \times IW + c_2 \times IW^2$$

Back

REFERENCES

1. S. Palacharla, N. Jouppi, and J. Smith, "Complexity-Effective Superscalar Processors", in Proceedings of the 24th International Symposium on Computer Architecture, June 1997.
2. S. Palacharla, N.P. Jouppi, and J.E. Smith, “Quantifying the Complexity of Superscalar Processors”, Technical Report CS-TR-96-1328, University of Wisconsin-Madison, November 1996.
3. K. C. Yeager, “MIPS R10000 Superscalar Microprocessor”, *IEEE Micro*, April 1996.
4. Linley Gwennap, “Intel’s P6 Uses Decoupled Superscalar Design” *Microprocessor Report*, 9(2), February 1995.
5. Eric Rotenberg, Steve Bennet, and J. E. Smith. “Trace Cache: a Low Latency Approach to High Bandwidth Instruction Fetching”, *Proccedings of the 29th Annual International Symposium on Microarchitecture*, December, 1996

REFERENCES

6. Keith I. Farkas, Norman P. Jouppi and Paul Chow. "*Register File Design Considerations in Dynamically Scheduled Processors*". In 2nd IEEE Symposium on High-Performance Computer Architecture, February 1996
7. T. Wada, S. Rajan, and S. A. Przybylski, “*An Analytical Access Time Model for On-Chip CacheMemories”*, IEEE Journal of Solid-State Circuits, 27(8):1147–1156, August 1992.
8. Steven J., E. Wilton and N. P. Jouppi, “*An Enhanced Access and Cycle Time Model for On-Chip Caches*” Technical Report 93/5, DEC Western Research Laboratory, July 1994.