Simple RISC instruction formats:
all instruction 32-bits in length

Arithmetic: add R1, R2, R3

opcode	dest reg	operand 1 reg	operand 2 reg	unused

Unconditional Branch/"jump": b someLabel

opcode	large offset from PC or absolute address

Arithmetic with immediate: addi R1, R2, 8

opcode	dest reg	operand 1 reg	immediate value

Conditional Branch: beq R1, R2, end_if

opcode	operand 1 reg	operand 2 reg	PC-relative offset to label

Load/Store: lw R1, 16(R2)

opcode	operand reg	base reg	offset from base reg

RISC Instruction Pipelining Example: One possible break down of instruction execution**.**

Stage	Abbreviation	Actions
Instruction Fetch	F	Read next instruction into CPU and increment PC by 4 byte (to next instruction)
Instruction Decode	D	Determine opcode, read registers, sign-extend immediate if needed, compute target address of all branch, update PC if unconditional branch
Execution / Effective addr	E	Calculate using operands prepared in D ▪ memory ref: add base reg to offset to form effective address ▪ reg-reg ALU: ALU performs specified calculation ▪ reg-immediate ALU: ALU performs specified calculation ▪ compare registers if condition branch and update PC if taken
Memory access	M	▪ load: read memory from effective address into pipeline register ▪ store: write reg value from ID stage to memory at effective address
Write-back	W	▪ ALU or load instruction: write result into register file

Branch Prediction - predict whether the branch will be taken and fetch accordingly

Static Techniques:
a) Predict never taken - continue to fetch sequentially. If the branch is not taken, then there is no wasted fetches.

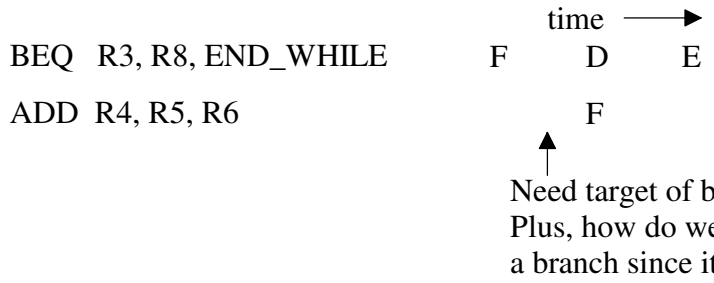
b) Predict always taken - fetch from branch target as soon as possible
(From analyzing program behavior, > 50% of branches are taken.)

c) Predict by opcode - compiler helps by having different opcodes based on likely outcome of the branch
Consider the HLL constructs:

HLL	**AL**
While (x > 0) do	BR_LE_PREDICT_NOT_TAKEN R3, #0, END_WHILE
{loop body}	
end while	END_WHILE:

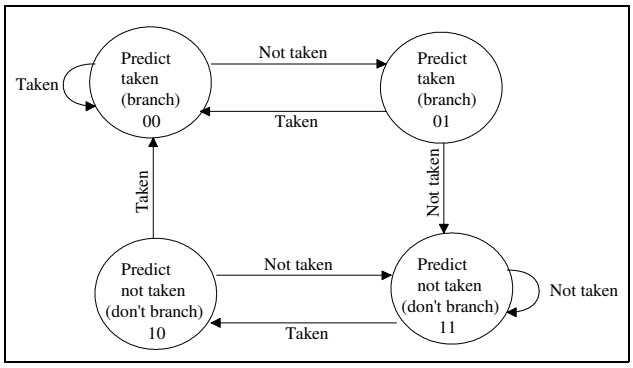
Studies have found about a 75% successful prediction rate using this technique.

Dynamic Techniques: try to improve prediction by recording program's history of conditional branch
Problem: How do we avoid always fetching the instruction after the branch?



Solution: Branch-prediction buffer (BPB)/Branch-History Table (BHT)- small, fully-associative cache to store information about most recently executed branch instructions. In a BPB, the Branch instruction address acts as the tag since that's what you know at F. During the F stage, the Branch-prediction buffer is checked to see if the instruction being fetched is a branch (if the addresses match) instruction.

Valid Bit	Branch Instruction Address (tag field)	Target Address of Branch	Prediction Bits	



If the instruction is a branch instruction and it is in the Branch-prediction buffer, then the target address and prediction can be supplied by the BPB **by the end of F for the branch instruction**.

If the branch instruction is in the Branch-prediction buffer, will the target address supplied correspond to the correct instruction to be execute next?

What if the instruction is a branch instruction and it is not in the Branch-prediction buffer?

Should the Branch-prediction buffer contain entries for unconditional as well as conditional branch instructions?

The table below shows the advantage of using a Branch-prediction buffer to improve accuracy of the branch prediction. It shows the impact of past ***n*** branches on prediction accuracy. Typically, two prediction bits are use so that two wrong predictions in a row are need to change the prediction

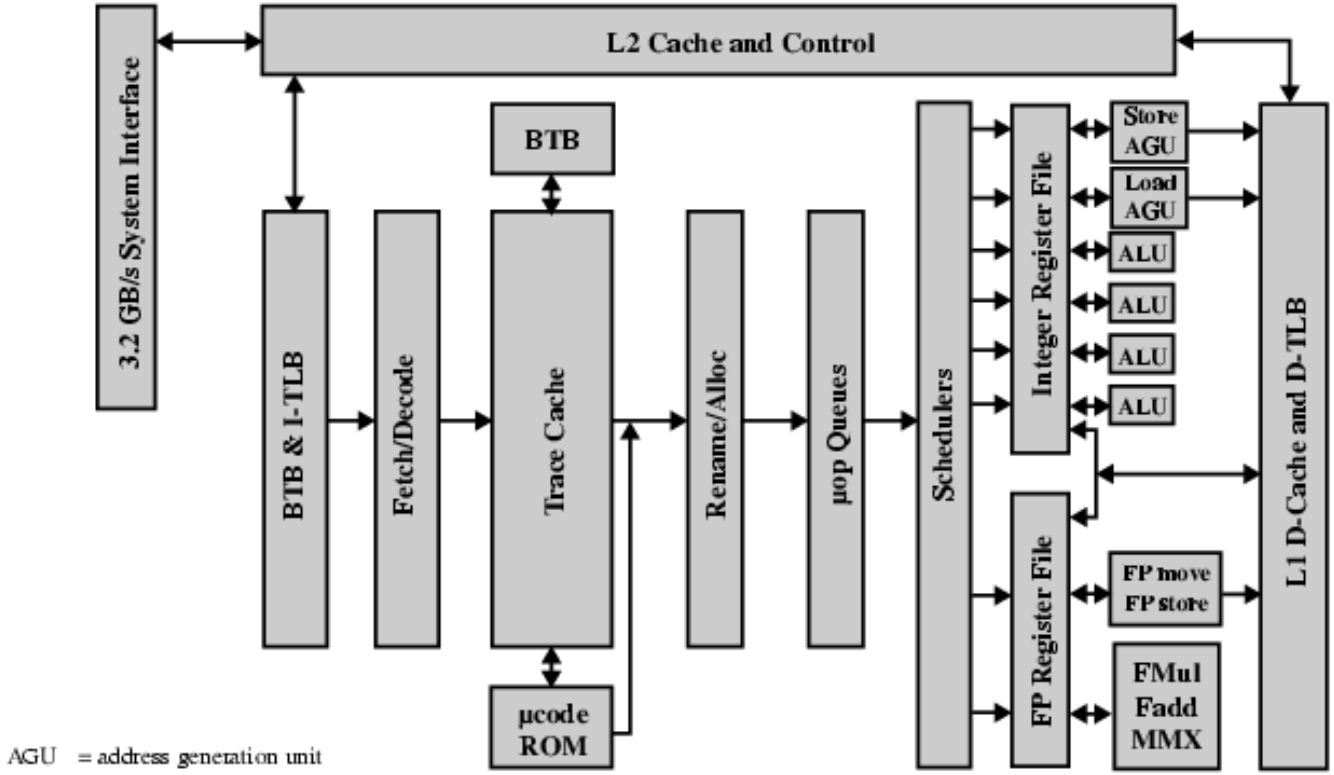
	Type of mix		
n	Compiler	Business	Scientific
0	64.1	64.4	70.4
1	91.9	95.2	86.6
2	93.3	96.5	90.8
3	93.7	96.6	91.0
4	94.5	96.8	91.8
5	94.7	97.0	92.0

Notice:

1) the big jump in using the knowledge of just 1 past branch to predict the branch
2) notice the big jump in going from using 1 to 2 past branches to predict the branch for scientific applications.
   What types of data do scientific applications spend most of their time processing?
   What would be true about the code for processing this type of data? .

Pentium 4 Processor

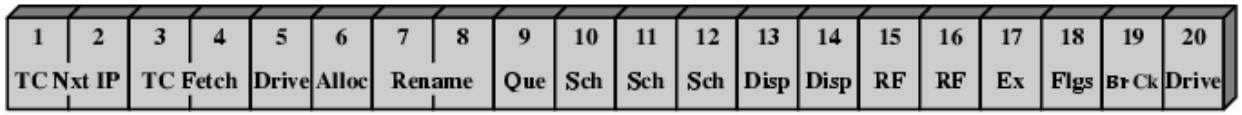
- 80486 - CISC
- Pentium
  - some superscalar components
  - two separate integer execution units
- Pentium Pro – Full blown superscalar
- Subsequent models refine & enhance superscalar design



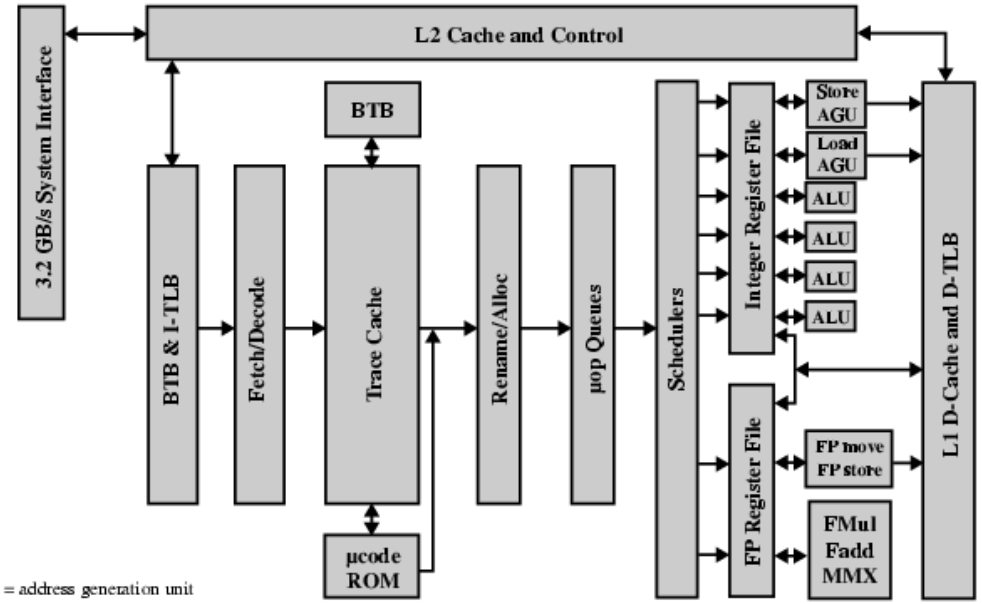
AGU = address generation unit
BTB = branch target buffer
D-TLB = data translation lookaside buffer
I-TLB = instruction translation lookaside buffer

Pentium 4 Operation:

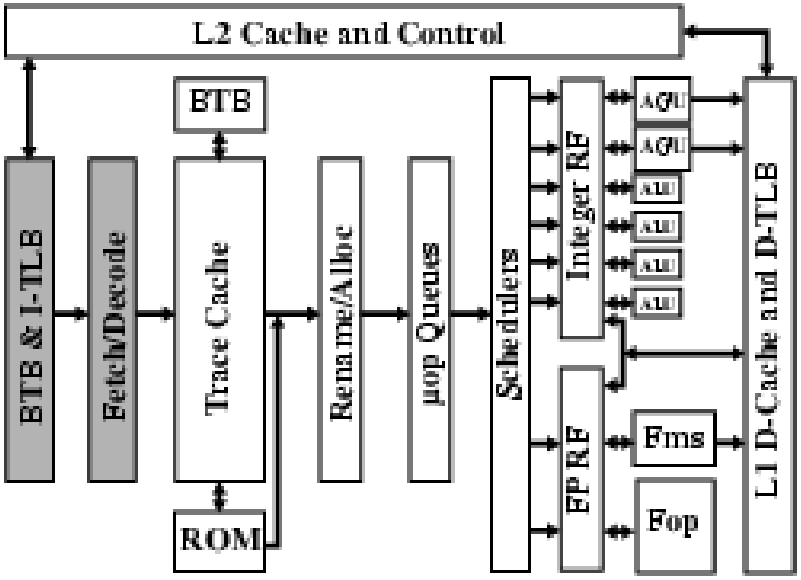
- Fetch x86 (CISC) instructions from memory in order of static program
- Translate each x86 instruction into one or more fixed length RISC instructions (micro-operations)
- Execute micro-ops on superscalar pipeline
  - micro-ops may be executed out of order
- Commit results of micro-ops to register set in original x86 program flow order
- Outer CISC shell with inner RISC core
- Inner RISC core pipeline at least 20 stages
  - Some micro-ops require multiple execution stages



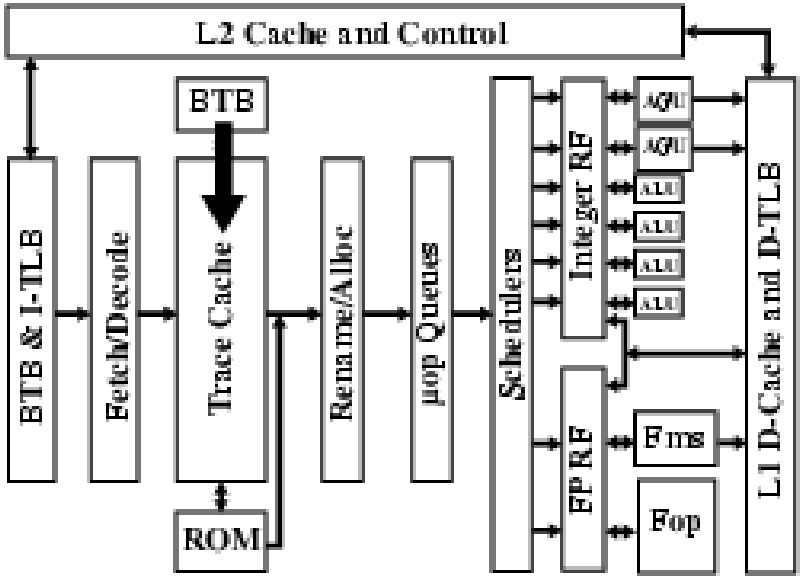
TC Next IP = trace cache next instruction pointer
TC Fetch = trace cache fetch
Alloc = allocate
Rename = register renaming
Que = micro-op queuing
Sch = micro-op scheduling
Disp = Dispatch
RF = register file
Ex = execute
Flgs = flags
Br Ck = branch check



AGU = address generation unit
BTB = branch target buffer
D-TLB = data translation lookaside buffer
I-TLB = instruction translation lookaside buffer

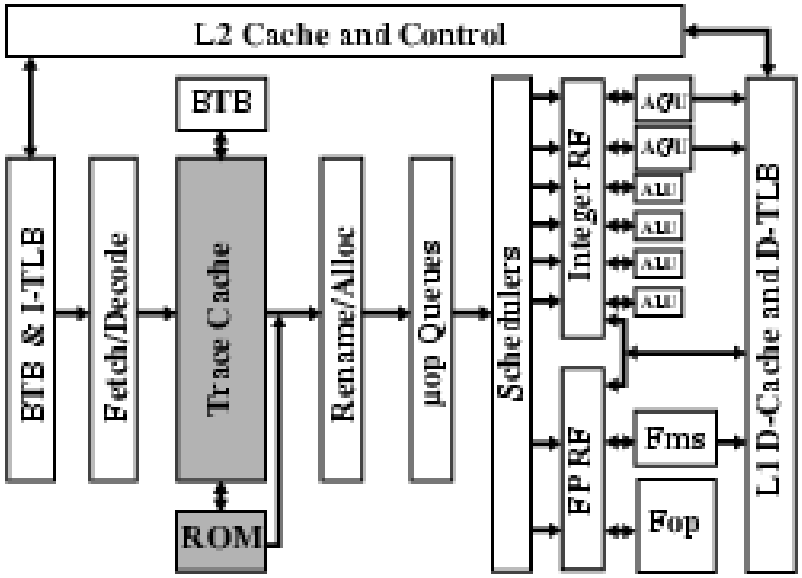


(a) Generation of micro-ops

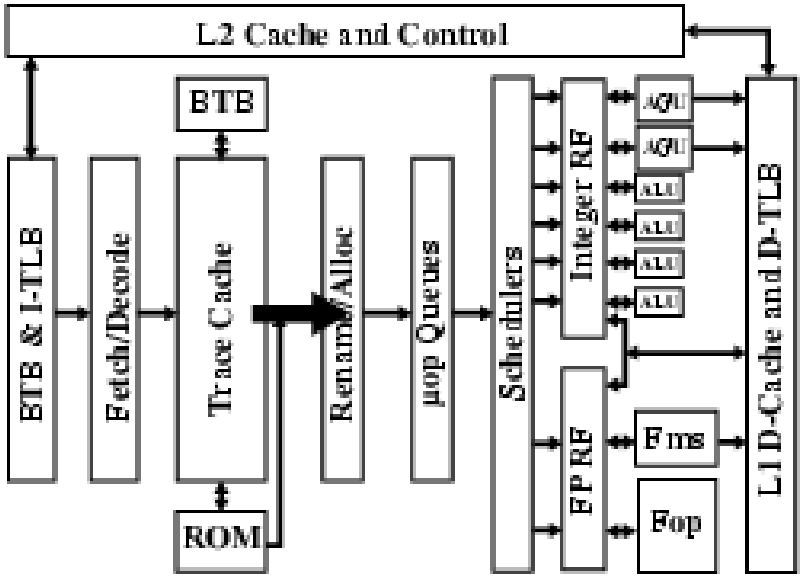


(b) Trace cache next instruction pointer

a) Fetch 64 bytes of Pentium 4 (CISC) instruction(s) from L2 cache and decode instruction boundaries and translates Pentium 4 (CISC) intructions into micro-op's (RISC)	b) Trace cache (L1 cache) stores recently executed mico-op's BTB uses dynamic branch prediction (a BHT) (4-bits used via Yeh's algorithm). Static prediction used if not in BTB.



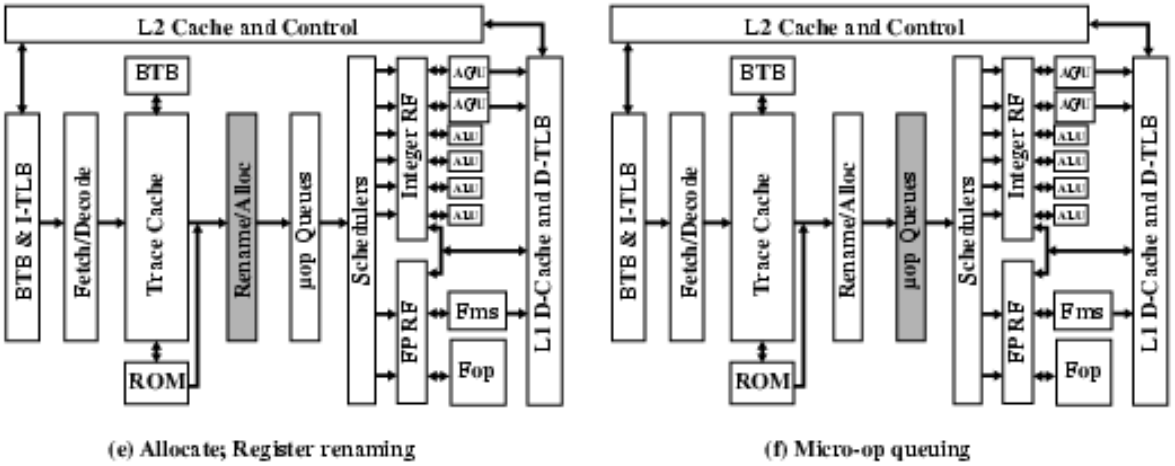
(c) Trace cache fetch



(d) Drive

c) Pulls micro-ops from cache (or ROM microprogrammed control unit for very complex instructions) in program sequence order	d) Drive delivers decoded instructions from the trace cache to the rename/allocate module.

Out-of-Order Execution Logic:

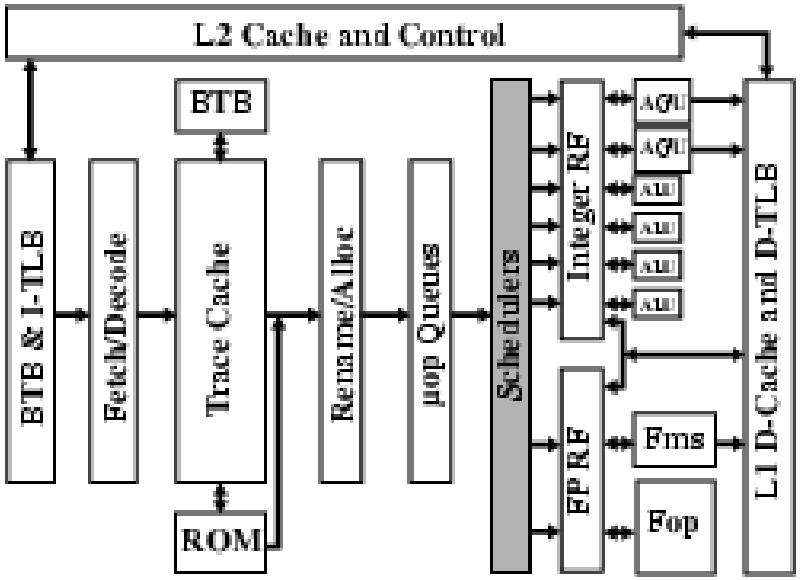


(e) Allocate; Register renaming

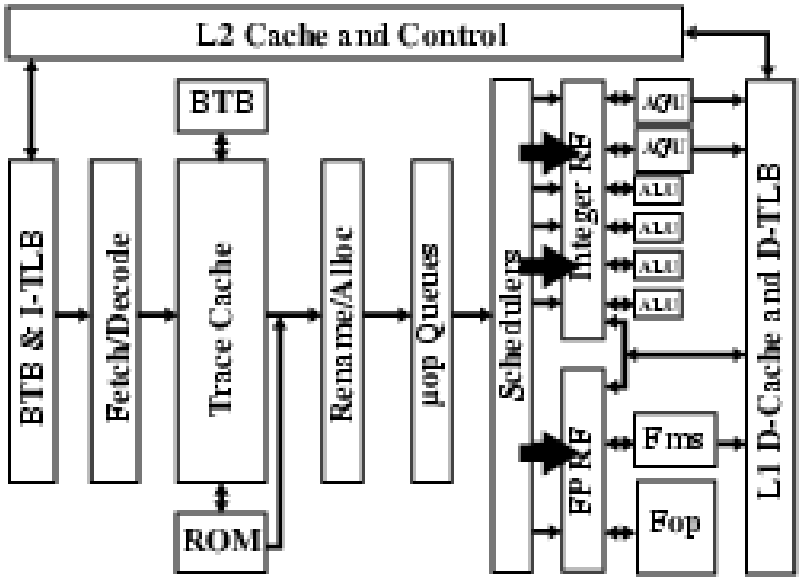
(f) Micro-op queuing

(ROB entry contains: state, memory address of generating instruction, micro-op, renamed register

Allocate - allocates resources needed for execution: • stalls pipeline if a resource (e.g., register) is unavailable • a reorder buffer (ROB) to store information about a micro-op as it executes • one of 128 integer or float registers for the result and/or one of 48 load buffers or one of 24 store buffers • an entry in one of the two micro-op queues	Two FIFO queues to hold micro-ops until there is room in the scheduler. One queue holds load or stores micro-ops One queue hold the remaining nonmemory micro-ops Queues can operate at different speeds

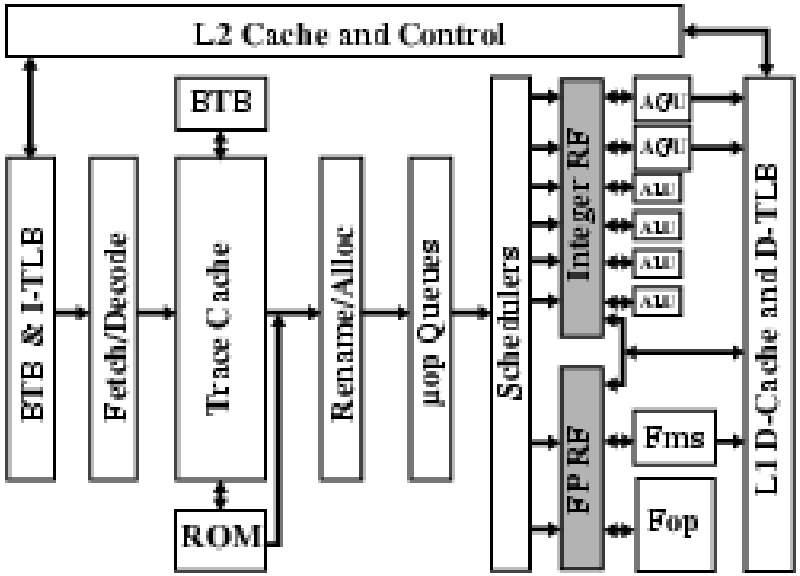


(g) Micro-op scheduling

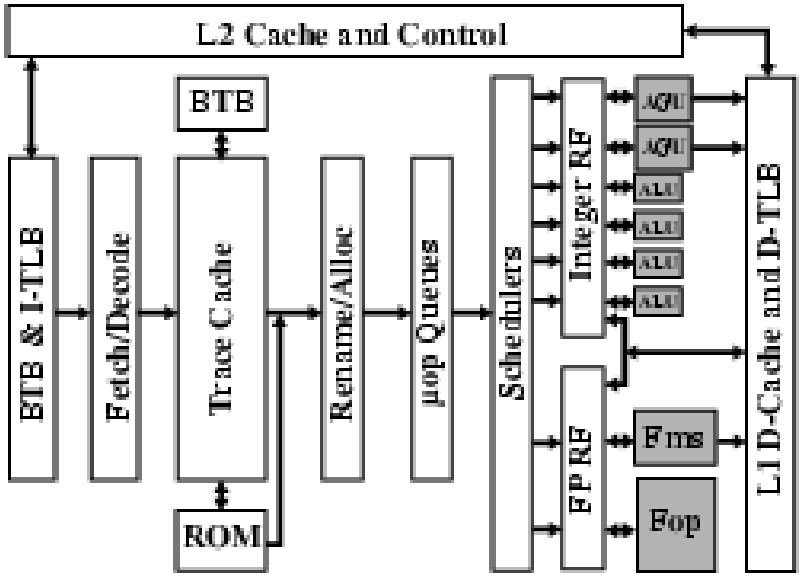


(h) Dispatch

Scheduler retrieves micro-ops from queues for dispatching/issuing for execution if all operands and execution unit are available.	Up to 6 micro-ops can be dispatched per cycle.

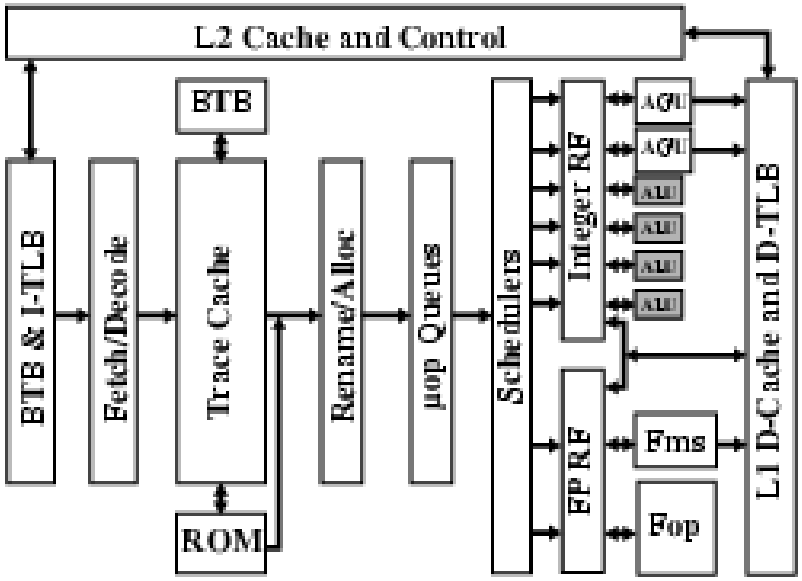


(i) Register file

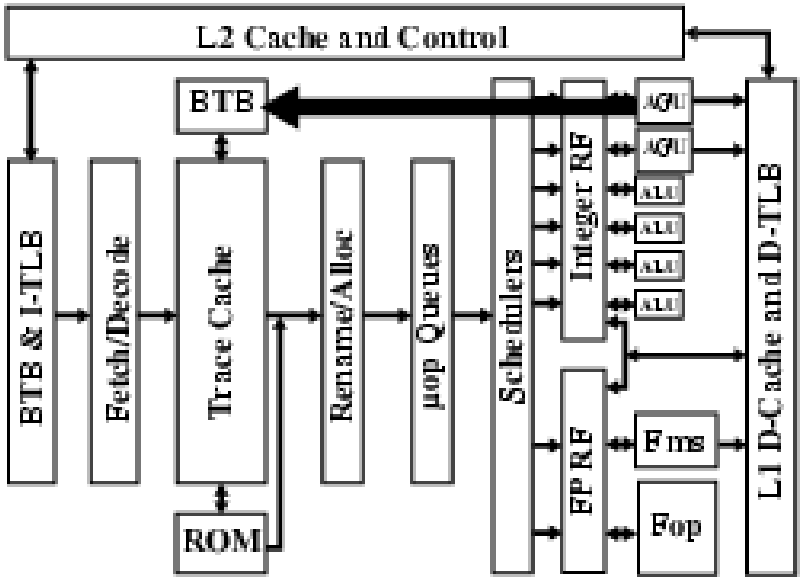


(j) Execute; flags

Execution units retrieve necessary integer and floating point registers	Compute flags - N, Z, C, V to use an input to the branches



(k) Branch check



(l) Branch check result

Compares the actual branch result with the prediction.	If branch outcome does not match prediction, remove micro-ops from the pipeline. Provide proper branch destination to the BTB which restarts the whole pipeline from the correct target address.