RISC/CISC Characteristics

(PowerPC) RISC Technology

References:
Chakravarty and Cannon, Chapter 2
Kacmarcik, Optimizing PowerPC Code

Modern programmers use assembly:
- for handcoding for speed
- for debugging

Common features of CISC:
- many instructions that access memory directly
- large number of addressing modes
- variable length instruction encoding
- support for misaligned accesses

Original goal of RISC (developed in the 1970's) was to create a machine (with a very fast clock cycle) that could process instructions at the rate of one instruction/machine cycle.

Pipelining was needed to achieve this instruction rate.

Typical current RISC chips are HP Precision Architecture, Sun SPARC, DEC Alpha, IBM Power, Motorola/IBM PowerPC

Common RISC characteristics

- <u>Load/store architecture</u> (also called register-register or RR architecture) which fetches operands and results indirectly from main memory through a lot of scalar registers. Other architecture is storage-storage or SS in which source operands and final results are retrieved directly from memory.
- <u>Fixed length instructions</u> which
  (a) are easier to decode than variable length instructions, and
  (b) use fast, inexpensive memory to execute a larger piece of code.
- <u>Hardwired controller instructions</u> (as opposed to microcoded instructions). This is where RISC really shines as hardware implementation of instructions is much faster and uses less silicon real estate than a microstore area.
- <u>Fused or compound instructions</u> which are heavily optimized for the most commonly used functions.
- <u>Pipelined implementations</u> with goal of executing one instruction (or more) per machine cycle.
- Large uniform register set
- minimal number of addressing modes
- no/minimal support for misaligned accesses

NOT NECESSARY for either RISC or CISC

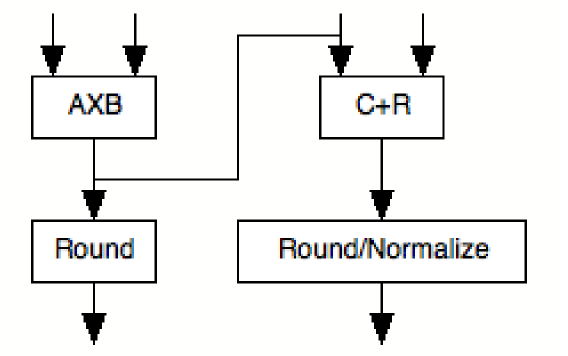
- instruction pipelining
- superscalar instruction dispatch
- hardwired or microcoded instructions

Fused instructions

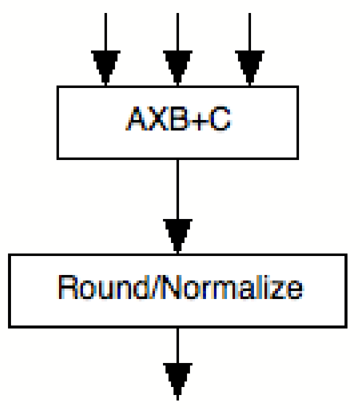
Classical FP multiply
1. Add exponents
2. Multiply significands
3. Normalize
4. Round off answer

Classical FP add
1. Subtract exponents
2. Align decimal points by shifting significand with smaller exponent to right to get same exponent
3. Add significands
4. Normalize
5. Round

Classical instruction



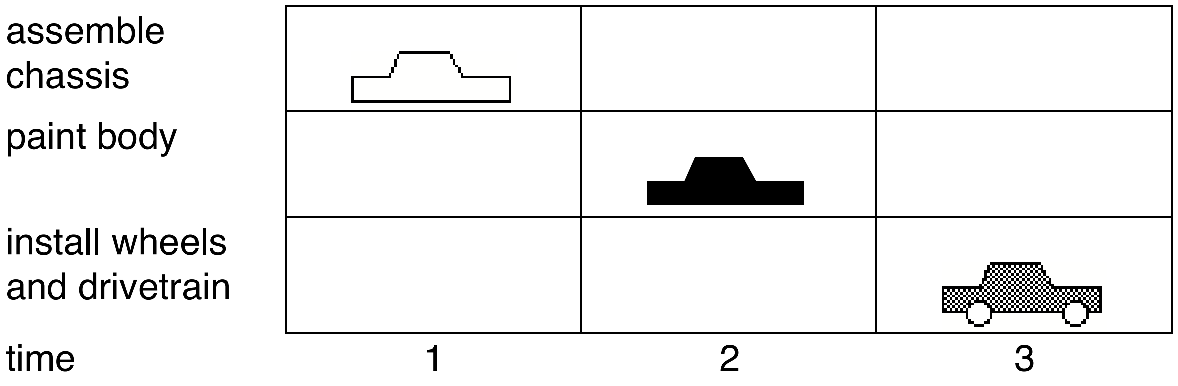
Fused instruction



PIPELINING

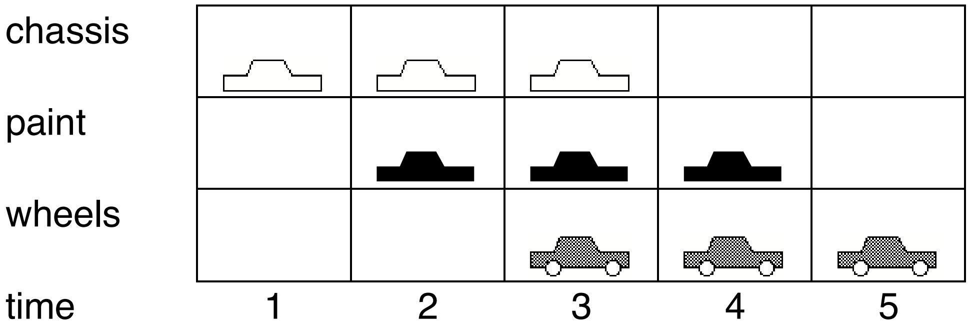
A conventional computer executes one instruction at a time with a Program Counter pointing to the instruction currently being executed. Pipelining is analogous to an oil pipeline where the last product may have gone in before the first result comes out. This provides a way to start a task before the first result appears. The computing throughput is now independent of the total processing time.

Conventional processing



A conventional process would require 9 time units to produce three cars.

Pipelined processing



A pipelined process would require 5 time units to produce the same number of cars.

INSTRUCTION PIPELINING

We can apply pipelining to the classical fetch/execute instruction processing. There are three phases to the fetch/execute cycle:

- instruction fetch
- instruction decode
- instruction execute

If we assume these all take one time unit (clock cycle) to execute a three stage pipeline will look like the following.

fetch	I_1	I_2	I_3	I_4	I_5	I_6
decode		I_1	I_2	I_3	I_4	I_5
execute			I_1	I_2	I_3	I_4
	time #1	time #2	time #3	time #4	time #5	time #6

Pipelining is great in theory but what if there is a branch in your code. You can't determine the next instruction to put into the pipeline until the branch instruction is executed. This can cause a hole, or "bubble" in the pipeline as shown below.

fetch	I_1	I_2			I_3	I_4	I_5
decode		I_1	I_2			I_3	I_4
execute			I_1	I_2			I_3
	time #1	time #2	time #3	time #4	time #5	time #6	time #6

Such bubbles represent performance degradation because the processor is not executing any instructions during this interval.

There are two techniques which can be used to handle this problem with branches:

- delayed branching (as done by an optimizing compiler)
- branch prediction (guess the result of the branch)

normal branch code	delayed branch code
instruction$_0$	instruction$_0$
instruction$_1$	instruction$_1$
instruction$_2$	branch*
branch	instruction$_2$
instruction$_3$	instruction$_3$
•	•
•	•
•	•
instruction$_n$	instruction$_n$

*Delay the instruction originally preceding the branch if it is does not influence the branch. This can be done by an optimizing compiler/assembler. The critical issue is how many independent instructions you have. This is a good technique for pipelines with a depth of 1-2 processes.

Branch prediction, on the other hand, works by "guessing" the target instruction for the branch and marking the instruction as a guess. If the guess was right then the processor just keeps executing; however, if the guess was wrong then the processor must purge the results. The key to this approach is a good guessing algorithm.

The PowerPC uses branch prediction. This approach is very good for FOR and DO/WHILE loops since the branch instruction always branches backwards <u>until</u> the final iteration of the loop. IF/THENs are very bad for guessing and are like flipping a coin with a 50% probability.

Probabilities of branch instructions:

instruction	probability of occurrence	probability of branch
unconditional branch (JMP)	1/3	1
loop closing (FOR and DO/WHILE, Dbcc, etc.)	1/3	~1
forward conditional branch (Bcc, etc.)	1/3	1/2

The forward conditional branches are the most difficult to guess. The worse case is that we will guess 1/3*1/2 of conditional branches wrong, causing bubbles about 50% of the time..

CISC/RISC tradeoffs

	RISC	CISC
general	very fast, fixed length instruction decode, high execution rate	fewer instructions, size of code is smaller
# of instructions	<100	>200
# of address modes	1-2	5-20
instruction formats	1-2	3+
average cycles/instruction	~1	3-10
memory access	load/store instructions only	most CPU instructions
registers	32+	2-16
control unit	hardwired	microcoded
instruction decode area (% of overall die area)	10%	>50%

RISC cycles
Performance of RISC machine comes from making optimum tradeoff between instruction set functionality (power of each instruction) and clock cycles/instruction.

Program_execution_time = num_instructions_executed * CPI * cycle_time

where num_instructions_executed is dependent upon the pipeline length, CPI is cycles/instruction, and cycle_time is 1/clock_frequency.

PowerPC (PPC)

This is a relatively new architecture with a lot of potential in technical applications.

PowerPC evolution

IBM POWER architecture	RS.9	1990	
	RS1	1990	
	RSC	1991	
PowerPC	601	1992	<--supports POWER instructions
	603	1993	
	604	1994	
		?	<--first 64 bit PPC’s

How does the PowerPC fit the RISC model?

- General purpose registers — 32 general purpose registers (any except GPR0 can be used as an argument to any instruction); 32 floating point registers
- LOAD/STORE architecture — only instructions that access memory are LOAD and STORE instructions
- Limited number of addressing modes
  - (I) register indirect;
  - (2) register indirect with register index;
  - (3) register indirect with immediate index.

  The branch instructions can be
  - (I) absolute; (2) PC relative; or (3) SPR (Special Purpose Register) indirect.
- Fixed length instructions — All PPC instructions are 32 bits long.
- No support for misalignments — RISC architecture should not allow misalignments to occur; however, POWER design considerations requiring emulation of other machines allows misalignments.

PPC Data Types

type	size (bits)	alignment
byte*	8	---------
half-word	16	--------0
word*	32	-------00
double word†	64	------000
quad word†	128	-----0000
floating point single*	32	-------00
floating point double*†	64	------000

* Most commonly used data types
†64 bit PPC implementation

Alignment

Address must be a multiple of data type size. Bytes are always aligned. Half words must be aligned to even bytes (multiples of 2) just like in the 68000; Words must be aligned to quad bytes (multiples of 4); etc.

Order of bytes

Big endian ordering of 0x0A0B

$0A	$0B

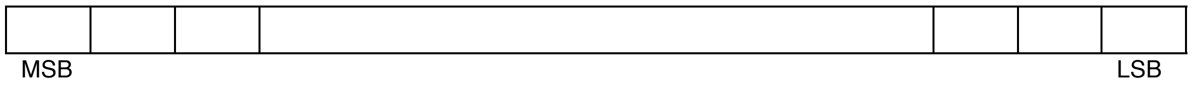
Little endian ordering of 0x0A0B

$0B	$0A

PPC and 68000 operate in bigendian mode. However, PPC has an option to switch modes.

Big endian ordering of bits in a register:

```
0    1    2    3                                         29   30   31
```


MSB
LSB

Super Scalar Implementation

<u>SuperScalar implementation (independent processing units)</u>
PPC 601 has 3 independent execution units so it can actually execute multiple instructions in a single clock cycle. Each execution unit is pipelined. PPC superscalar architecture can execute up to 5 operations/clock cycle.

There are currently two envisioned PPC architectures: 32 and 64 bit. Only the 32 bit implementations have been produced. The PPC architecture does NOT include any i/o definitions.

PPC registers are all 32 bits long (except floating point which are 64 bits long)

PPC consists of three independent processing units

1. branch processing unit	handles branch instructions
2. fixed point unit	also called instruction unit
3. floating point unit	does only floating point instructions

There are three classes of instructions to match the processing units:
1. branch
2. fixed point
3. floating point

All these instructions are 32 bits long and MUST be word aligned.

Because of the Load/Store architecture all computations MUST be done in registers as the operands MUST be loaded into registers BEFORE they can be manipulated/operated on. This typically requires a lot of registers.

PPC Registers

Branch processing unit has three main registers:

Link register	**LR**	contains return address from subroutine calls; contains target address for a branch*

*subroutines can return with a Branch_to_LR instruction

Count register	**CTR**	used for counting loop iterations; treats as Dbcc instructions significantly increasing performance
Counter register	**CTR**	holds number of iterations or a loop; can be used as the final count or as a decrementation counter

Condition register **CR** is the PPC status register

Condition register has 8 4-bit wide condition code fields.

0 3	4 7	8 11	12 15	16 19	20 23	24 27	28 31
CR0	CR1	CR2	CR3	CR4	CR5	CR6	CR7

These fields can be specified as a DESTINATION for results of a comparison, or as a SOURCE for conditional branches.

CR0 is usually used for fixed point comparisons

LT	GT	EQ	SO

where SO is the summary overflow. A summary overflow is a “sticky” overflow bit that remains set until reset.

CR1 is usually used for floating point comparisons

FL	FG	FR	FU

FL - floating point less than
FG -floating point greater than
FR - floating point equal
FP - floating point unordered

Fixed point operations with record bit

LT	GT	EQ	SO

LT - negative (<0)
GT -positive (>0)
EQ - zero (=0)
SO - summary overflow

Floating point operations with record bit

FX	FEX	VX	OX

FX - floating point exception summary
FEX -floating point enabled exception summary
VX - floating point invalid operation exception summary
OX - floating point overflow exception

Fixed point processor has most used registers:

32 general purpose registers	**GPR0 - GPR31**	32 bits wide in 32 bit implementations; 64 bits wide in 64 bit implementations; used for all data storage and fixed point operations
Exception register	**XER**	carry, overflow, byte count, and comparison for string instructions

SUPERVISOR MODE REGISTERS:

Machine state register	**MSR**	is processor in 32 or 64 bit mode; are interrupts enabled; bigendian vs. little endian mode
Save/Restore registers	**SSRn**	indicate machine status when an interrupt occurs plus information required to restore state after an interrupt
Processor verification register	**PVR**	READ ONLY. Processor version information.

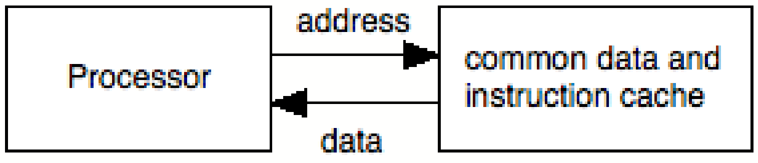
PLUS LOTS MORE!

Floating point processor is similar to fixed point processor:

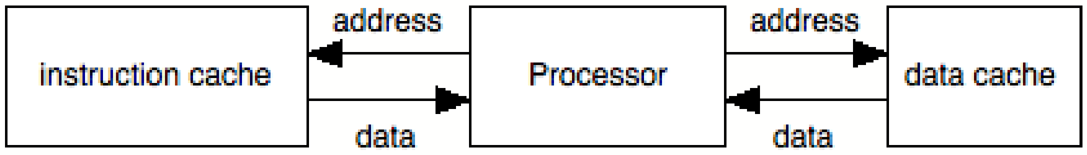
32 floating point registers	**FPR0 - FPR31**	64 bits wide in all implementations; 64 bit registers which are the source and destination for all floating point operations
Floating point status and control register	**FPSCR**	handles floating point exceptions and status of floating point operations; enable bits for fp exceptions; rounding bits to control rounding; status bits to record fp exceptions

PPC Architecture

Many RISC processors use a Harvard architecture; the 601 uses a von Neumann architecture.



von Neumann architecture



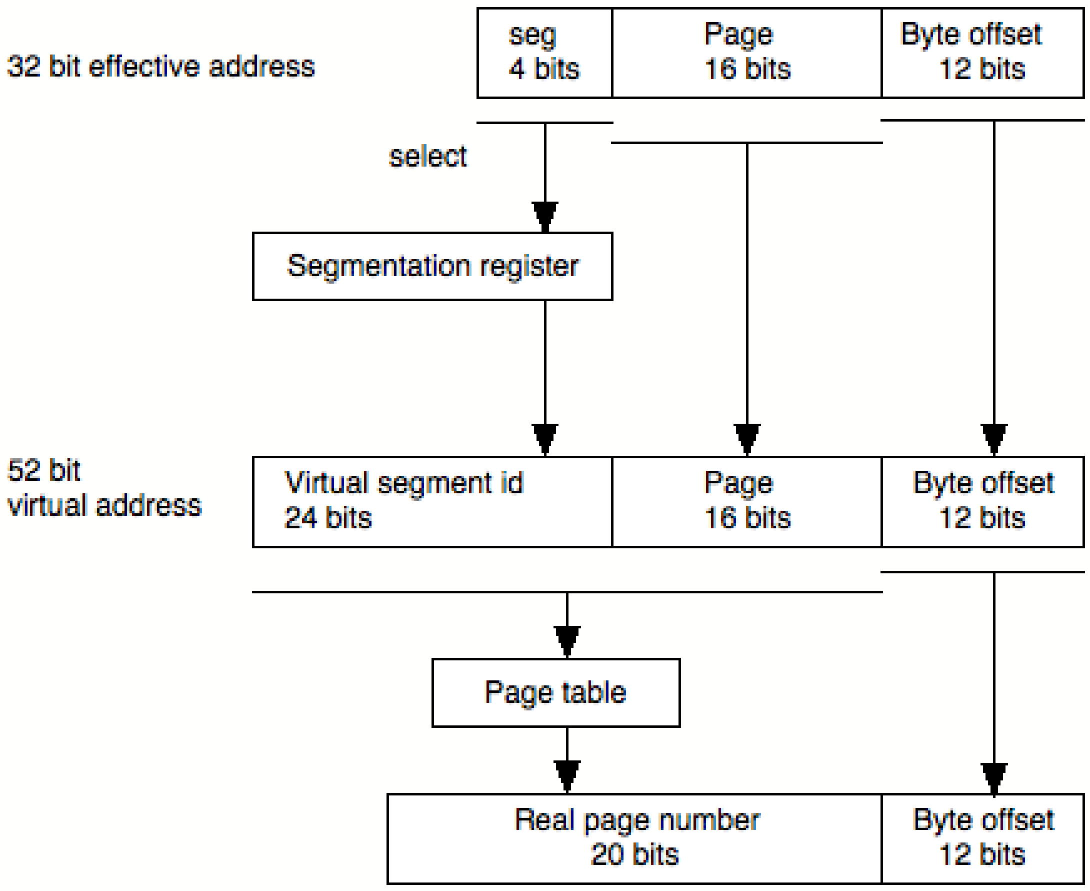
Harvard architecture

Address translation

Effective addresses on the PPC must be translated before they can actually access a physical location. Block address translation takes precedence.

segmented address translation		i/o address
	virtual address	i/o address
		physical address
block address translation	real address	
	i/o address	

Segmented addressing:



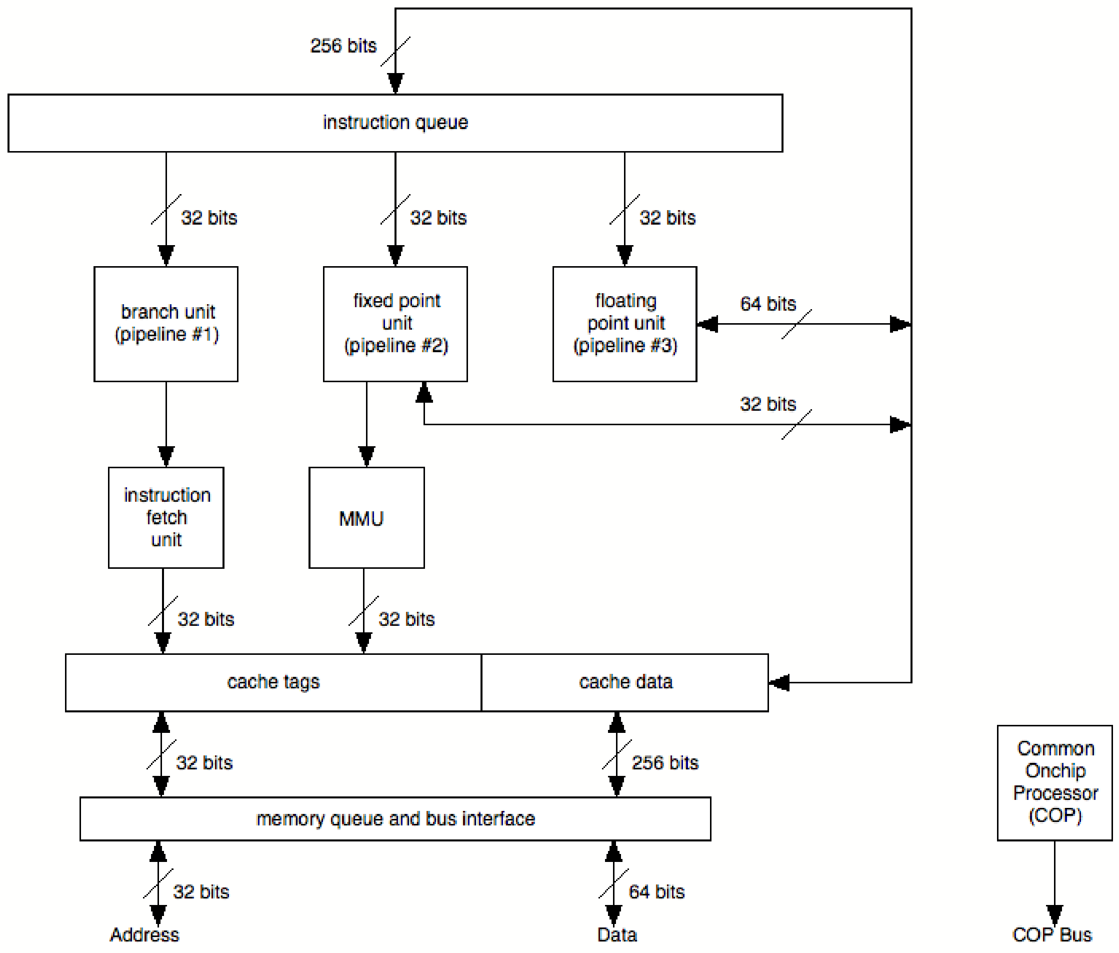
64 bit implementation is VERY different.

Block addressing:

Paged addressing using 4k pages. Block consists of at least 32 pages 128kB up to 65536 pages (256 MB).

The PPC also contains a 64 bit time base register and a 32 bit decrement register which can be used for timing.

Overall PPC 601 architecture:



NOTE: The COP processor controls built-in self test, debug and test features at boot time.

CACHE

Cache is a small memory that acts as a buffer between the processor and main memory. On-chip cache access times are typically 1-2 clock cycles long; access of regular external memory is typically much longer, perhaps 20-30 clock cycles long.

Basic principle of a cache
Locality of reference - Whenever a program refers to a memory address there are likely to be more references to nearby addresses shortly thereafter.

Way cache works
Whenever the main program references a memory location a block of memory containing the referenced address is copied to the cache. The idea is that a lot of following instructions will use information from this cache dramatically speeding up the performance of the processor.

How good a cache works in speeding up computation depends upon:
1. the design of the cache
2. the nature of the executing code

Cache Design and Organization in the PPC

Processor	Size Instruction/ Data	Associativity	Replacement Policy	Line Size (bytes)
601	32K unified	8	LRU	32/64
603	8K/8K	2/2	LRU	32
604	16K/16K	4/4	LRU	32
620	32K/32K	8/8	LRU	64

Notes:

Cache line	the block of memory in the cache that holds the loaded data
Cache tag	pointer from a cache line to the main memory
Line Size	the number of bytes associated with a tag
Associativity	relationship between main memory and the cache. If any block from the main memory can be loaded into any line of the cache, the cache is fully associative. More performance is usually obtained by limiting the number of lines into which a block might reside - this occurs because you have a smaller number of places to look for a particular address. In a two-way associative memory the cache controllerwould only have to examine two tags; in a 4 way four tags; and in an 8-way right tags.
Replacement	When the processor is loading a new block to cache and all the potential lines are full, the cache controller will replace an occupied line with the new data. Common replacement schemes are: first-in first-out (FIFO), least recently used (LRU), and random,
Writeback	(versus store-through) Refers to how the cache contoller handles updates to the information in the cache. In astore-through scheme the stored data is immediately posted to both the cache and main memory. In a store-in (or writeback) scheme only the information in the cache is updated immediately

(the line is marked *dirty*) and only updated when the line is replaced in the cache.

Power PC family:

The PPC 601 has three pipelines:

Pipeline #1 (2 stage)	Fetch	Dispatch Decode Execute Predict				
Pipeline #2 (3 stage)	Fetch	Dispatch Decode	Execute	Writeback		
or						
Pipeline #2 (4 stage)	Fetch	Dispatch Decode	Address Generation	Cache (optional)	Writeback	
Pipeline #3 (6 stage)	Fetch	Dispatch	Decode	Execute1 (Multiply)	Execute2 (Add)	Writeback

*writeback i(or store-in caching) is what happens when the PPC updates data in the cache; posting to main memory is delayed until the line is replaced by the cache unit.

The 603 was designed for portable applications and has four pipelines
1. branch processing unit (2 stage pipeline)
2. fixed point unit (3 stage pipeline)
3. floating point unit (6 stage pipeline)
4. load/store unit (5 stage pipeline)

It also has dynamic power management which controls the processor clock so that only units in use are power up.

The 604 was designed for desktop applications and has two additional integer units giving much improved integer performance. It is in a 304 pin ceramic flat pack with 3.6 million transistors. It dissipates 10 watts at 100 MHz and is based upon 0.5μm CMOS technology.

The embedded versions (4xx, EC403, EC401, etc.) are probably the most economically important.

MAJOR PPC INSTRUCTION GROUPS

- Branch and trap
- Load and store
- Integer
- Rotate and shift
- Floating point
- System integer

Can add suffixes in [ ] to modify instructions

Integer instructions
[o] update FP Exception Register XER
[.] record condition information in CR0

Floating point
[s] single precision data
[.] record condition information in CR1

Branch
[l] (all instructions) record address of following instruction in link register
[a] (some instructions) specified address is absolute

Branch instructions

b[l][a] addr	unconditional branch
b[l][a] BO,BI.addr	conditional branch

[a] indicates that target address is absolute
BO indicates the branch on condition (nine bits and can get complicated)
BI specifies which bit of the CR register is to be used in the test

NOTE: PPC assemblers use b instead of %
Example:
b0000y which indicates decrement the CTR and branch if CTR≠0 and CR[BI]=0.

The y bit encodes hints as to whether branch is likely to be taken.

bcctr [l] BO,BI	branch conditional to count register

Often used to count in loops.

bclr [l] BO,BI	branch conditional to link register

Used for returning from subroutines.

There are probably at least 8-12 extended versions of each basic branch instruction.

lbz rT,d(rA)	load byte and zero; displacement with respect to contents of a source register
load	load from memory location into target register
load with update	add offset afterwards
load indexed	calculate address from two registers
load indexed with update	combination of above
store	write contents of register to memory
store with update	
store with indexed	
store indexed with update	

Example function that performs 64-bit integer addition

```
#   Struct unsigned64
#   {
#      unsigned hi;
#      unsigned lo;
#   }
#
#   unsigned64 add64(unsigned64 *a, unsigned64 *b);
#
# Expects
#   r3 pointer to struct unsigned64 result
#   r4 pointer to unsigned64a
#   r5 pointer to struct unsigned64b
#
# Uses
#   r3 pointer to result
#   r4 pointer to a
#   r5 pointer to b
#   r6 high order word of a (a.hi), high word of sum
#   r7 low order word of a (a.lo), low word of sum
#   r8 high order word of b (b.hi)
#   r9 low order word of b (b.lo)

        lwz     r7,4(r4)        #r7<--a.lo - load word and zero
                                load the word (words are 32 bits on
                                the PPC) into r7, uses r4 as its
                                source
        lwz     r9,4(r5)        #r9<--b.lo - load word and zero
        lwz     r6,0(r4)        #r6<--a.hi load word and zero
        addc    r7,r7,r9        #r7<--sum lo, set CA - add carrying
        lwz     r8,0(r5)        #r8<--b.hi - load word and zero
        stw     r7,4(r3)        #result.lo <--r7 - store word
        adde    r6,r6,r8        #r6<--sum hi with CA - add extended
        stw     r6,0(r3)        #result hi <--r6 - store word
        blr                     #return - branch to link
```