

# Contents Lecture 6

- Computer architecture development
- Pipelined processors
- More about the IBM Power architecture
- Superscalar processors

# The earliest computers

- The earliest computers e.g. Z3, Z4, UNIVAC I, EDSAC and EDSAC had very little hardware
- One register called the *ackumulator* was the destination of all ALU operations
- The first general purpose machine with a register file was built 1956 in England: the Pegasus which had eight registers.

# Stack architectures

- Code size has always been important and one way to reduce code size is to load and store implicit operands on the stack.
- Current examples include some HP calculators and the Java virtual machine.
- Stack architectures can also simplify compilation, to a limited extent.
- Burroughs produced several commercial machines in the early 1960 which were stack architectures.
- The Burroughs machines had e.g. special instructions for managing the call frames to better support Algol.
- Stack machines fell out of fashion in the 1970s.

# The first computer architecture: IBM 360

- In the early 1960s, when a new machine was introduced, it normally was binary incompatible with other machines, which means software is thrown away.
- At that time IBM sold seven different and incompatible machines which led to increased costs.
- In 1964 the term **computer architecture** was introduced by IBM when it designed the IBM 360, a family of compatible machines with different performance and costs. This was a major breakthrough.
- IBM 360 machines were byte-addressable and had general purpose registers, and was a huge success.

# The IBM z series

- IBM mainframes are now called the z series for zero downtime
- IBM z14 is an IBM 360 compatible machine launched 2017.
- It has 10-core processors clocked at 5.2 GHz and can have up to 32 TB of RAM memory.
- Not all but many applications from the 1970's can run unmodified on it.

# Pipelined computers

- Pipelining is a technique to make e.g. the assembly of cars more efficient.
- The key is that by having *simple* steps, each step can be fast.
- We will see more about this but recall that instruction execution has the steps:
  - 1 Instruction fetch
  - 2 Instruction decode and operand fetch
  - 3 Execute
  - 4 Memory access
  - 5 Write back
- The insight was that we might be able to have five instructions executing concurrently, one in each step, leading to a five times faster machine.

- Thornton and Cray and others at CDC were the first to explore pipelined instruction execution
- Their CDC 6600 was the first pipelined load-store architecture.
- The CDC 6600 was basically a RISC machine (we will get back to that)
- The CDC 6600 designers realized the essential connection (later forgotten) between a "clean architecture" and the possibility of an efficient pipelined implementation of it.
- This clean type of architecture came back in the 1980's with Power, MIPS, SPARC, Alpha, 88000, and later the ARM and many others.

# A Software Crisis and the Semantic Gap

- In the 1960's, hardware costs were huge compared with software costs.
- This switched in the early 1970's and more and more complex software started to become painfully expensive.
- Some people thought the solution was to design machines which would simplify compilation of high-level languages, and a so called **semantic gap** was identified (in fact it was only nonsense).
- For example, Digital Equipment designed the VAX architecture as a true memory-to-memory architecture, which means that an instruction can fetch operands from memory and write the result back to memory.
- Examples of fancy VAX instructions: polynomial evaluation and stack frame control.



- Why were computers being designed with more and more complex instructions?
- One reason was the time of instruction fetching: smaller code size leads to fewer instruction cache misses and faster code.
- A load-store architecture such as the CDC 6600, all operands must be fetched from memory using load instructions, and written back using a store instruction.
- Another reason was that product lines competed with having more complex instructions.
- Recall the French engineer, author and pilot Antoine de Saint-Exupéry: *"An engineer knows that he is ready, not when there is nothing more to add but when there is nothing more to take away."*

# The IBM 801

- John Cocke and his group of researchers at IBM Yorktown Heights started in 1979 a project to design a new architecture from scratch without any garbage from previous product lines remaining as constraints:
- Load-store architecture
- No complex instructions that reduce the clock frequency
- 16 general purpose registers (later changed to 32 registers)
- Intended to be used only with high level languages and an optimizing compiler
- This was the RISC concept (although the term was coined at Berkeley)
- Intel tried the same with their IA-64 Itanium but it was too late to replace X86 for them — AMD saw their opportunity to make a 64-bit X86 and then Intel could not afford to miss that market.

# The Power Architecture

- The IBM 801 was a research prototype and was commercialized as the IBM Power architecture
- The POWER1 from 1990 had 800,000 transistors and was a superscalar machine in which several instructions could *start* executing concurrently.
- The POWER2 had 15 million transistors per chip and was released 1993. It was used in Deep Blue which beat chess world champion Garry Kasparov 1997.

- The POWER4 had 174 million transistors and was released 2001 and was the first chip multiprocessor.
- POWER8 was released in 2014 has up to 12 cores per chip and 8 hardware threads per core (96 threads per chip), and is clocked at up to 5 GHz.
- The `power.cs.lth.se` machine is a POWER8 machine with ten cores, clocked at 3.5 GHz.
- POWER10 was released in August 2020
  - 15 cores,
  - added more support for efficient matrix computations for AI workloads
  - can address up to 2 petabytes of RAM

# The Power Architecture

- Registers
- Instruction formats
- Classes of instructions:
  - Branch instructions
  - Fixed point instructions
  - Floating point instructions
  - Vector instructions
  - Support for encryption
- Program examples

# Power Registers

- 32 general purpose registers, 32/64 bits  
R0 means zero for some instructions
- 32 floating point registers, 64 bits
- 32 vector registers, 128 bits
- a number of special purpose registers, such as for:
  - storing loop iteration count to avoid  $i++$  and  $i < n$ .
  - function call return address
- Fixed point exception register XER, 32 bits  
Three bits hold: summary overflow, overflow, and carry. The summary overflow is set when overflow is set but only cleared explicitly by writing to XER.

# Power Registers: Link register

- Functions start with a prologue which sets up the stack frame and ends with an epilogue which removes it.
- 32/64 bits
- The link register holds the return address for function calls and is written implicitly: `bl printf /* function call. */` by the branch-and-link instruction.
- Accessed as a Special Purpose Register: SPR 8.
- Reading the link register: `mfspr R0, 8`, or `mflr R0` in the prologue.
- Writing the link register: `mtspr R0, 8`, or `mtlr R0` in the epilogue.

# Power Registers: Count register

- This is a special purpose register

- Count register, 32/64 bits

The count register can be used to control loop termination, which is faster than using general purpose registers, compare, and branch. Only for one inner loop.

```
for (i = low; i < high; ++i)
    S;
```

```
/* low in R3, high in R4. */
sub    5,4,3 /* R5 = high - low. */
mtctr 5      /* Repeat R5 times. */
L: S        /* Statement S. */
bdnz   L     /* Decrement and branch to L if nonzero.*/
```



# Power Registers: 8 four-bit condition registers

- Four fields in each register:
  - Bit 0: Negative
  - Bit 1: Positive
  - Bit 2: Zero (or equal)
  - Bit 3: Summary overflow
- Fixed point instructions optionally set CR0 (bits 0..3).
- Floating point instructions optionally set CR1 (bits 4..7).
- There are move and logical instructions for operating on the condition registers.

# Power Registers: more registers

- 32 floating point registers, 64 bits
- Conforms to IEC 60559, i.e. the IEEE 754 floating point standard.
- 32 vector registers, 128 bits

# Power Instruction Formats 1(2)

- An instruction format defines what the instruction bits mean. The Power has several formats and the more commonly used include:
- I-form: e.g. for function call

18	LI
6	26

- B-form: for conditional branches

16	BO	BI	BD	AA	LK
6	5	5	16	1	1

# Power Instruction Formats 2(2)

- D-form: e.g. for  $RT = RA + D$ .

PO	RT	RA	D
6	5	5	16

- X-form: e.g. for  $RT = RA + RB$ .

PO	RT	RA	RB	XO	LK
6	5	5	5	10	1

How can the processor know which format to use?  
The primary opcode (PO) gives this information.

# Integer Numbers

- C/C++ and other languages support both signed and unsigned integers.
- Signed numbers are represented in two's-complement form.
- A 4-bit register can represent the unsigned values 0..15 and the signed values -8..7.
- Positive numbers are represented in normal binary form, e.g. 5 as 0101.
- Negative numbers are represented as 16-X, e.g. -5 as  $10000 - 00101 = 01111 + 00001 - 00101 = 01010 + 00001 = 01011 = 1011$ .
- We then expect that  $-5 + 5$  would be zero:  
 $1011 + 0101 = 10000 \mod 2^{16} = 0000$

*Q: What does  $1000_2$  mean? A: Either  $-8$  or  $+8$ . It depends on the data type!*

# Power Compare Instructions

- Four integer compare instructions:
  - Signed compare, register-register
  - Signed compare, register-signextended immediate
  - Unsigned compare, register-register
  - Unsigned compare, register-immediate
- Destination is a condition register field (any of the eight).
- By default the assembler uses CR field 0 for integer compare.
- By sign-extended is meant the most significant bit of the 16 bit constant is copied to the other bits to preserve the value.
- Signextend -8 from 4 to 6 bits: 1000 becomes 111000 since it means  $-32 + 16 + 8 = -8$

**There are five main branch instructions:**

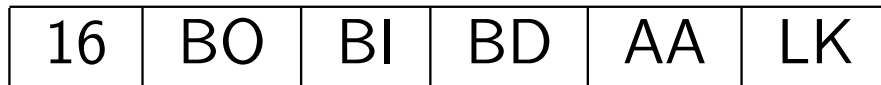
- Unconditional branch I-form, used for function calls.
- Conditional branch B-form, very powerful branch instruction.
- Branch conditional to Link Register, used for function return.
- Branch conditional to Count Register, can be used.
- System call (switch from user to operating system kernel).

# Branch I-form

- Six bits opcode 18 and 24 bits branch offset LI,
- one bit AA: if AA then  $NIA = \text{EXTS}(LI)$  else  $NIA = CIA + \text{EXTS}(LI)$ ,
- one bit LK: if LK then  $LR = CIA$  (used for function call),
- Where NIA means Next instruction address (new value of PC),
- CIA means Current instruction address (address of this instruction),
- EXTS means extend sign,
- and LR means Link register.



# Power Branch B-form



- Six bit opcode 16, 5 bit branch options (see next slide),
- 5 bits BI select one bit in the condition register (ie, from any CR field)
- BD is branch offset and AA and LK the same as for branch I-form.

# Power Branch Options

BO	Description
0000y	CTR-=1; branch if CTR != 0 and COND is false
0001y	CTR-=1; branch if CTR == 0 and COND is false
001zy	branch if COND is false
0100y	CTR-=1; branch if CTR != 0 and COND is true
0101y	CTR-=1; branch if CTR == 0 and COND is true
011zy	branch if COND is true
1z00y	CTR-=1; branch if CTR != 0
1z01y	CTR-=1; branch if CTR == 0
1z1zz	branch always

- COND is bit selected by BI and z is don't care.
- y is a branch-prediction hint made by the programmer or compiler.

# Examples of Power Branch Instructions

Branch instruction	Extended mnemonic	Description
bc 16,0,L	bdnz L	Decrement and branch if nonzero
bc 4,2,L	bne L	branch if CR0 reflects "not equal"
bc 4,14,L	bne L	branch if CR3 reflects "not equal"

- Some disassemblers show the extended mnemonics
- As assembler programmers it is preferable to use the extended mnemonics

# Power Arithmetic Instructions

- Many arithmetic instructions *optionally* can set the three first bits of the CR0 register (i.e. negative, positive, or zero) using the Rc bit of the instruction, and overflow (in the fourth bit of CR0 and in XER) using the OE bit.
- In assembler, these are specified with a dot and o as suffixes: addo . 1,2,3 means  $R1 = R2 + R3$ ; One of  $<$ ,  $>$ ,  $=$  and overflow are stored in CR0 bits 0..3.
- The D-form arithmetic instructions have a 16 bit constant which for most instructions is signextended.
- Eg addi uses the value 0 instead of the contents of register 0.

# Power Fixed Point Add Instructions 1(2)

Add instruction	Description/Real instruction	"Informal" Name
<code>addi rt,ra,si</code>	$rt = (ra == 0 ? 0 : (ra)) + \text{exts}(si)$	Add Immediate
<code>li rt,si</code>	<code>addi rt,0,si</code>	Load Immediate
<code>subi rt,ra,si</code>	<code>addi rt,ra,-si</code>	Subtract Immediate
<code>addis rt,ra,si</code>	$rt = (ra == 0 ? 0 : (ra)) + \text{exts}(si) \parallel 0x0000$	Add Immediate Shifted
<code>lis rt,si</code>	<code>addis rt,0,si</code>	Load Immediate Shifted
<code>addis rt,ra,si</code>	$rt = (ra == 0 ? 0 : (ra)) + \text{exts}(si) \parallel 0x0000$	Add Immediate Shifted
<code>add rt,ra,rb</code>	$rt = (ra) + (rb)$	Add
<code>add. rt,ra,rb</code>	$rt = (ra) + (rb); \text{set CR0}$	Add
<code>addo rt,ra,rb</code>	$rt = (ra) + (rb); \text{set OV,SO}$	Add
<code>addo. rt,ra,rb</code>	$rt = (ra) + (rb); \text{set OV,SO,CR0}$	Add
<code>addic. rt,ra,si</code>	$rt = (ra) + \text{exts}(si); \text{set CA,CR0}$	Add Immediate Carrying
<code>addc rt,ra,rb</code>	$rt = (ra) + (rb); \text{set CA}$	Add Carrying
<code>addc. rt,ra,rb</code>	$rt = (ra) + (rb); \text{set CA,CR0}$	Add Carrying

# Power Fixed Point Add Instructions 2(3)

Add instruction	Description/Real instruction	"Informal" Name
addco rt,ra,rb	$rt = (ra) + (rb); \text{set } CA, OV, SO$	Add Carrying
addco. rt,ra,rb	$rt = (ra) + (rb); \text{set } CA, OV, SO, CR0$	Add Carrying
adde rt,ra,rb	$rt = (ra) + (rb) + CA; \text{set } CA$	Add Extended
adde. rt,ra,rb	$rt = (ra) + (rb) + CA; \text{set } CA, CR0$	Add Extended
addeo rt,ra,rb	$rt = (ra) + (rb) + CA; \text{set } CA, OV, SO$	Add Extended
addeo. rt,ra,rb	$rt = (ra) + (rb) + CA; \text{set } CA, OV, SO, CR0$	Add Extended
addme rt,ra	$rt = (ra) + CA - 1; \text{set } CA$	Add to Minus One
addme. rt,ra	$rt = (ra) + CA - 1; \text{set } CA, CR0$	Add to Minus One
addmeo rt,ra	$rt = (ra) + CA - 1; \text{set } CA, OV, SO$	Add to Minus One
addmeo. rt,ra	$rt = (ra) + CA - 1; \text{set } CA, OV, SO, CR0$	Add to Minus One

# Power Fixed Point Add Instructions 3(3)

Add instruction	Description/Real instruction	"Informal" Name
addze rt,ra	rt = (ra) + CA; set CA	Add to Zero Extended
addze. rt,ra	rt = (ra) + CA; set CA,CR0	Add to Zero Extended
addzeo rt,ra	rt = (ra) + CA; set CA,OV,SO	Add to Zero Extended
addzeo. rt,ra	rt = (ra) + CA; set CA,OV,SO,CR0	Add to Zero Extended

- Researchers at IBM have found that their compilers and/or assembler programmers can make good use of these instructions.
- This is still a *Reduced* Instruction Set Architecture
- Better name is: Set of Reduced Instructions.
- Of course you don't need to learn all these instruction but you should get the picture about what's in the Power

# Example Power Logical Instructions 1(2)

Instruction		Description	Name
andi.	rt,ra,ui	$rt = (ra) \& UI$ ; set CR0	And Immediate
andis.	rt,ra,ui	$rt = (ra) \& (ui \ll 0x0000)$ ;set CR0	And Immediate Shifted
ori	rt,ra,ui	$rt = (ra) \mid UI$	Or Immediate
oris	rt,ra,ui	$rt = (ra) \mid (ui \ll 0x0000)$	Or Immediate Shifted
xori	rt,ra,ui	$rt = (ra) \wedge UI$	Xor Immediate
xoris	rt,ra,ui	$rt = (ra) \wedge (ui \ll 0x0000)$	Xor Immediate Shifted
and	rt,ra,rb	$rt = (ra) \& (rb)$	And
and.	rt,ra,rb	$rt = (ra) \& (rb)$ ;set CR0	And
or	rt,ra,rb	$rt = (ra) \& (rb)$	Or
or.	rt,ra,rb	$rt = (ra) \& (rb)$ ;set CR0	Or
xor	rt,ra,rb	$rt = (ra) \& (rb)$	Xor
xor.	rt,ra,rb	$rt = (ra) \& (rb)$ ;set CR0	Xor



# Example Power Logical Instructions 2(2)

Instruction		Description	Name
nand	rt,ra,rb	$rt = \neg((ra) \& (rb))$	Nand
nand.	rt,ra,rb	$rt = \neg((ra) \& (rb)); \text{set CR0}$	Nand
nor	rt,ra,rb	$rt = \neg((ra) \mid (rb))$	Or
nor.	rt,ra,rb	$rt = \neg((ra) \mid (rb)); \text{set CR0}$	Or
xor	rt,ra,rb	$rt = (ra) \& (rb)$	Xor
xor.	rt,ra,rb	$rt = (ra) \& (rb); \text{set CR0}$	Xor
eqv	rt,ra,rb	$rt = (ra) \wedge \neg (rb)$	Equivalent
eqv.	rt,ra,rb	$rt = (ra) \wedge \neg (rb); \text{set CR0}$	Equivalent
andc	rt,ra,rb	$rt = (ra) \& \neg (rb)$	And complement
andc.	rt,ra,rb	$rt = (ra) \& \neg (rb); \text{set CR0}$	And complement
orc	rt,ra,rb	$rt = (ra) \mid \neg (rb)$	Or complement
orc.	rt,ra,rb	$rt = (ra) \mid \neg (rb); \text{set CR0}$	Or complement

# Power Memory Access Instructions 1(4)

Instruction		Description	Name
lbz	rt,d(ra)	$rt = M[(ra == 0 ? 0 : (ra)) + \text{exts}(d)]$	Load Byte and Zero
lbzx	rt,ra,rb	$rt = M[(ra == 0 ? 0 : (ra)) + (rb)]$	... Indexed
lbzu	rt,d(ra)	$ea = (ra) + \text{exts}(d); rt = M[ea]; ra = ea$	... with Update
lbzux	rt,ra,rb	$ea = (ra) + (rb); rt = M[ea]; ra = ea$	... Indexed with Update
lhz	rt,d(ra)	$rt = M[(ra == 0 ? 0 : (ra)) + \text{exts}(d)]$	Load Halfword and Zero
lhzx	rt,ra,rb	$rt = M[(ra == 0 ? 0 : (ra)) + (rb)]$	... Indexed
lhzu	rt,d(ra)	$ea = (ra) + \text{exts}(d); rt = M[ea]; ra = ea$	... with Update
lhzux	rt,ra,rb	$ea = (ra) + (rb); rt = M[ea]; ra = ea$	... Indexed with Update
lha	rt,d(ra)	$rt = \text{exts}(M[(ra == 0 ? 0 : (ra)) + \text{exts}(d)])$	Load Halfword Algebraic
lhax	rt,ra,rb	$rt = \text{exts}(M[(ra == 0 ? 0 : (ra)) + (rb)])$	... Indexed
lhau	rt,d(ra)	$ea = (ra) + \text{exts}(d); rt = \text{exts}(M[ea]); ra = ea$	... with Update
lhaux	rt,ra,rb	$ea = (ra) + (rb); rt = \text{exts}(M[ea]); ra = ea$	... Indexed with Update

# Power Memory Access Instructions 2(4)

Instruction		Description	Name
lwz	rt,d(ra)	$rt = M[(ra == 0 ? 0 : (ra)) + \text{exts}(d)]$	Load Word and Zero
lwzx	rt,ra,rb	$rt = M[(ra == 0 ? 0 : (ra)) + (rb)]$	... Indexed
lwzu	rt,d(ra)	$ea = (ra) + \text{exts}(d); rt = M[ea]; ra = ea$	... with Update
lwzux	rt,ra,rb	$ea = (ra) + (rb); rt = M[ea]; ra = ea$	... Indexed with Update
lwa	rt,d(ra)	$rt = \text{exts}(M[(ra == 0 ? 0 : (ra)) + \text{exts}(d)])$	Load Word Algebraic
lwax	rt,ra,rb	$rt = \text{exts}(M[(ra == 0 ? 0 : (ra)) + (rb)])$	... Indexed
lwau	rt,d(ra)	$ea = (ra) + \text{exts}(d); rt = \text{exts}(M[ea]); ra = ea$	... with Update
lwaux	rt,ra,rb	$ea = (ra) + (rb); rt = \text{exts}(M[ea]); ra = ea$	... Indexed with Update
ld	rt,d(ra)	$rt = M[(ra == 0 ? 0 : (ra)) + \text{exts}(d)]$	Load Doubleword
ldx	rt,ra,rb	$rt = M[(ra == 0 ? 0 : (ra)) + (rb)]$	... Indexed
ldu	rt,d(ra)	$ea = (ra) + \text{exts}(d); rt = M[ea]; ra = ea$	... with Update
ldux	rt,ra,rb	$ea = (ra) + (rb); rt = M[ea]; ra = ea$	... Indexed with Update
lmw	rt,d(ra)	for r in rt..31 lwz r	Load Multiple Word

# Power Memory Access Instructions 3(4)

Instruction		Description	Name
stb	rs,d(ra)	$M[(ra == 0 ? 0 : (ra)) + \text{exts}(d)] = rs$	Store Byte
stbx	rs,ra,rb	$M[(ra == 0 ? 0 : (ra)) + (rb)] = rs$	... Indexed
stbu	rs,d(ra)	$ea = (ra) + \text{exts}(d); M[ea] = rs; ra = ea$	... with Update
stbux	rs,ra,rb	$ea = (ra) + (rb); M[ea] = rs; ra = ea$	... Indexed with Update
sth	rs,d(ra)	$M[(ra == 0 ? 0 : (ra)) + \text{exts}(d)] = rs$	Store Halfword
sthx	rs,ra,rb	$M[(ra == 0 ? 0 : (ra)) + (rb)] = rs$	... Indexed
sthu	rs,d(ra)	$ea = (ra) + \text{exts}(d); M[ea] = rs; ra = ea$	... with Update
sthux	rs,ra,rb	$ea = (ra) + (rb); M[ea] = rs; ra = ea$	... Indexed with Update

# Power Memory Access Instructions 4(4)

Instruction		Description	Name
stw	rs,d(ra)	$M[(ra==0?0:(ra))+exts(d)]=rs$	Store Word
stwx	rs,ra,rb	$M[(ra==0?0:(ra))+(rb)]=rs$	... Indexed
stwu	rs,d(ra)	$ea=(ra)+exts(d); M[ea]=rs; ra=ea$	... with Update
stwux	rs,ra,rb	$ea=(ra)+(rb); M[ea]=rs; ra=ea$	... Indexed with Update
std	rs,d(ra)	$M[(ra==0?0:(ra))+exts(d)]=rs$	Store Doubleword
stdx	rs,ra,rb	$M[(ra==0?0:(ra))+(rb)]=rs$	... Indexed
stdu	rs,d(ra)	$ea=(ra)+exts(d); M[ea]=rs; ra=ea$	... with Update
stdux	rs,ra,rb	$ea=(ra)+(rb); M[ea]=rs; ra=ea$	... Indexed with Update
stmw	rs,d(ra)	for r in rs..31 stw r	Store Multiple Word

# Power Function Call Conventions

- The function call conventions specify:
  - Which register is the stack pointer?
  - Which register is the frame pointer (if any) ?
  - How should parameters and return values be passed?
  - Where is the return value saved?
- This specification is for a particular combination of processor and *operating system*
- Even though Linux/Power and MacOS X/Power use identical hardware, and quite similar call conventions, there are some important differences.
- In this course we will not go into the details of the call conventions but you need to understand the disassembled Power code on Linux to some extent.

# Linux/Power Function Call Convention Basics

- The stack grows toward lower addresses.
- Register R1 is the stack pointer and its value is always 16 byte aligned (the value in R1 is a multiple of 16).
- Callee may destroy R0, R2 - R12, F0 - F13, V0-V19, LR, CTR, CR0, CR1, CR5-CR7. The others must be saved and restored by the callee.
- Integer parameters are passed in R3..R10 and remaining parameters in are copied to the *caller's* stack frame.
- Return values are passed in R3.
- Small struct/union parameters are copied to the callee as any other parameters.
- For larger struct/unions the address is used and if it is modified in the called function it is copied there in order not to break the copy semantics for parameters in C/C++.

# Example Function and Effects of Compiler Optimization

.L.sum:

```
std 31,-8(1)
stdu 1,-64(1)
mr 31,1
mr 9,3
mr 0,4
stw 9,112(31)
stw 0,120(31)
lwz 9,112(31)
lwz 0,120(31)
add 0,9,0
extsw 0,0
mr 3,0
addi 1,31,64
ld 31,-8(1)
blr
```

```
int add(int a, int b)
{
    return a + b;
}
```

.L.sum:

```
add 3,3,4
extsw 3,3
blr
```

```
// extsw = extend signed word.
// extsw makes sure r3 is in the
// range of the type int, by
// copying bit 31 to bits 32..63.
```



# Adding 64-bit Integers

```
long long sum(long long a, long long b)
{
    return a + b;
}
```

// 32-bit mode

```
sum:    addc 10,6,4
        adde 9,5,3
        mr 4,10
        mr 3,9
        blr
```

// 64-bit mode

```
sum:    add 3,3,4
        blr
```

- In 32-bit mode:  $a = r_3 \times 2^{32} + r_4$  and  $b = r_5 \times 2^{32} + r_6$ .
- If there is a carry bit coming out to the left when adding  $r_6$  and  $r_4$  it is used as input when adding  $r_5$  and  $r_3$  due to `addc` and `adde`.
- Don't forget to use the `-m64` switch to `gcc`!

# Example C programs translated to Power: WHILE

```
int f(int n)
{
    int    i;
    int    s;

    s = 0;
    i = 0;

    while (i < n) {
        s += i;
        i += 1;
    }
    return s;
}
```

gcc with optimization:

```
f:
    mr. 0,3
    li 9,0      # i = 0
    li 3,0      # s = 0
    blelr- 0
    mtctr 0
.L6:
    add 3,3,9
    addi 9,9,1
    bdnz .L6
    blr
```

- `mr.` copies  $n$  to  $r_0$
- The dot in `mr.` request that it should be checked whether  $r_0$  becomes less, equal or greater than zero.
- `blelr-` is conditional return if  $n \leq 0$
- The minus in `blelr-` says the branch is unlikely which helps the processor to guess what to do next.

# Recall the stages in instruction execution

- Fetch: read an instruction from memory
- Decode: interpret what the bits mean and read operands from register file
- Execute: perform an ALU operation, or calculate a memory address
- Memory access: only for load and store instructions
- Write back: write back result to a register

*These five steps are an example. Some processors have 20 steps.*

# Another Example Power Assembler Program

```
add    r3,r4,r5    ; R3 = R4 + R5
subf   r6,r7,r8    ; R6 = R8 - R7
addi   r6,r6,1     ; R6 = R6 + 1
lwz    r7,4(r5)    ; R7 = MEM[R5 + 4]
add    r6,r6,r7    ; R6 = R6 + R7
```

# Instruction Execution in the Five-Stage Pipeline

Clock cycle	FETCH	DECODE	EXECUTE	MEMORY ACCESS	WRITE BACK
1	add				
2		add			
3			add		
4				add	
5					add
6	subf				
7		subf			
8			subf		
9				subf	
10					subf

*The add and subf instructions take ten clock cycles to execute.*

# Pipelined Execution

Clock cycle	FETCH	DECODE	EXECUTE	MEMORY ACCESS	WRITE BACK
1	add				
2	subf	add			
3	addi	subf	add		
4	lwz	addi	subf	add	
5	add	lwz	addi	subf	add
6		add	lwz	addi	subf
7			add	lwz	addi
8				add	lwz
9					add

*The best we can do is completing one instruction per clock cycle.*

# True Data dependencies

- The `subf` produces a value which the `addi` consumes, and the `lwz` produces a value which the last `add` consumes.
- Since an instruction writes back a value to the register at the last pipeline stage, the value read in the second stage is not up-to-date.
- This is called a *True Dependence* or *Read-after-write hazard* (RAW).
- The dependence between the `subf` and the `addi` can be handled by adding more hardware to "forward" the result to the `addi`.
- Forwarding is not possible from the memory access to the execute stages

# Output dependencies

- If two instructions write to the same register, there is an *Output dependence* between them (or *Write-after-write hazard*).

```
div.    r4,r5,r6      ; modifies R4
subf    r4,r8,r9      ; also modifies R4 so output dependent
                        ; on the div instruction
```

- If the `div.` takes a lot of time, we don't want the `subf` to wait.
- Superscalar processors solve this in hardware using *register renaming*.



# Anti dependencies

- If one instruction reads a register and a subsequent writes to it, there is an *Anti dependence* between them (or *write-after-read hazard*).

```
add    r4,r5,r6    ; modifies R4
stw    r4,16(r6)    ; reads R4 and saves R4 at M[16 + R6]
subf   r4,r8,r9     ; modifies R4 so anti
                        ; dependent on the stw
```

If the `stw` takes a lot of time, we don't want the `subf` to wait.

# Pipeline Stalls

- Since the `lwz` gets the data from memory at the end of the fourth pipeline stage and the data is needed at the beginning of third stage, we must suffer a one cycle delay.
- This is controlled by the hardware which stops the execution of the instruction which depends on the `lwz` (all subsequent instructions are also stalled).
- Other stalls happen at branch instructions. If the processor uses the ALU to calculate the new address for the PC, it has nothing to do for a few cycles until the new PC is used to fetch instructions.
- Modern processors use hardware which tries to guess early where each branch is going and then speculatively fetch instructions from there. This helps a lot.

# Reducing the Effects of Pipeline Stalls

- What can programmers do about pipeline stalls?
  - Avoid using global variables in inner loops.
  - Avoid using virtual functions — but the project's main goal might not be reducing pipeline stalls but rather deliver a product on time. Be wise.
  - Avoid using many branches in inner loops.
- What can compilers do about pipeline stalls? A lot:
  - By finding which instructions depend on which, compilers try to *schedule* instructions so a producer instruction executes sufficiently ahead of the consumer instructions.
  - Allocating global variables to registers in the loop, so you don't have to.

# Instruction Latency and Throughput

- The *latency* of an instruction is the number of cycles it takes to produce the result.
- The latency is *not* reduced by pipelining.
- Throughput is the number of instructions (of a certain kind) the processor can complete per cycle once the pipeline has been filled.
- For example: a pipelined floating point add may have a latency of five clock cycles and a throughput of one: *with no true dependences*, one add can complete every cycle. An integer add takes one cycle and is not pipelined (except for fetch/decode/execute...).
- Usually the divide instruction is not pipelined. The latency may be 30 cycles and the throughput 1/30.

# Superscalar processors

- A pipelined processor as we just saw was state-of-the-art for workstations during the 1980's and is typical for some processors for embedded systems today.
- Current high-performance processors try to complete multiple instructions every clock cycle. Our power.cs.1th.se can have more than 200 instructions executing in each of the ten cores.
- For instance, several instructions may be sent to different parts of the core every clock cycle.
- A superscalar processor has multiple so called functional units, eg two single-cycle integer ALUs, two pipelined floating-point units, two pipelined vector units, at least one load-store unit, and a special branch processing unit.

# Essential features in a superscalar processor

- Speculative execution: instructions can start execute before it is known that they really should, but they are **not** permitted to permanently modify (destroy) either memory or registers.
- Three essential features of a superscalar processor are:
  - **Branch prediction:** hardware fetches instructions from memory where it guesses the program will go. Usually they predict the right way. When a misprediction is detected, all wrong-path instructions must be marked as such.
  - **Reorder buffer:** every instruction is put in a FIFO queue and they may only update "state" (e.g. memory) if they reach the end of the FIFO and have not been killed.
  - **Register renaming:** a technique to remove output and anti-dependencies at the hardware level.
- These three together make it possible to execute instructions speculatively. A speculatively executed instruction can modify a rename register but not memory. If it is cancelled, the new register value in the rename register is simply not copied to the real register.

# Branch prediction

- The processor has tables where previous branch outcomes are stored.
- When fetching an instruction at address  $A$ , the processor checks the tables and decided whether the next instruction to fetch is at  $A + 4$  or an address stored in the table.
- When "looking" at a superscalar processor, one can see that it sometimes can start fetch **and execute** instructions in a called method before the call instruction has executed!
- Such instructions are executed speculatively and it must be easy for the hardware to cancel them if needed for some reason (not for this course: e.g. a pagefault occurs before the call).

# Register renaming 1(4)

- Anti and output dependences at the register level are not real dependences
- They exist because some instructions happened to use the same register number for different purposes.
- Instructions with anti and output dependences do not communicate data between them.
- In a true dependence, one instruction really needs a value computed by some other instruction, so it **must** wait until the value has been computed.
- Register renaming at the hardware level removes anti and output dependences.



## Register renaming 2(4)

- Consider register renaming for the integer registers, called the general purpose registers (GPR) on the Power.
- There are 32 GPRs and e.g. 12 rename registers for these.
- There is a data structure (in hardware, of course) which says in which rename register the most recent value of each GPR is.
- When an instruction wants to *read* a register, e.g. R3, it checks the data structure to see if the normal R3 is up-to-date, or if the value is in a rename register and which.
- When an instruction wants to *write* to a register, it asks for a new rename register, but if none is available the instruction must wait.

# Register renaming 3(4)

- With this scheme we can have 12 instructions in the pipeline which modify integer registers.
- Whether the modify the same or different integer registers does not matter.
- What if an instruction wants a value from a rename register, but that value is still being computed, i.e. not yet finished?
- Then a so called tag (or ticket) for that rename register is given to the instruction who wants to read the register.
- When the rename register is updated, the instructions with a tag waiting for that register can proceed.

# Register renaming 4(4)

- Not all registers are renamed.
- Typically on a Power, the integer, floating point, vector, and condition registers are renamed.
- On some Power processors earlier than ours, the link register was not renamed.

# Reorder buffer

- The Reorder buffer is a FIFO and controls that all instructions finish in the program order (unless it is certain that they cannot "make troubles": a simple add cannot but a store or conditional branch can).
- When this FIFO is full no new instructions can be sent to the various functional units (integer ALU, floating pointer ALU, etc).
- If an instruction is to be cancelled, then the data structure for the rename registers must be updated, so that no future instruction gets the value produced from a cancelled instruction.
- Some instructions are so called *serialized* which means they are slower, e.g. by letting all previous instructions leave the FIFO before starting. Eg extended add (with carry) and move to/from link or condition registers can be serialized.