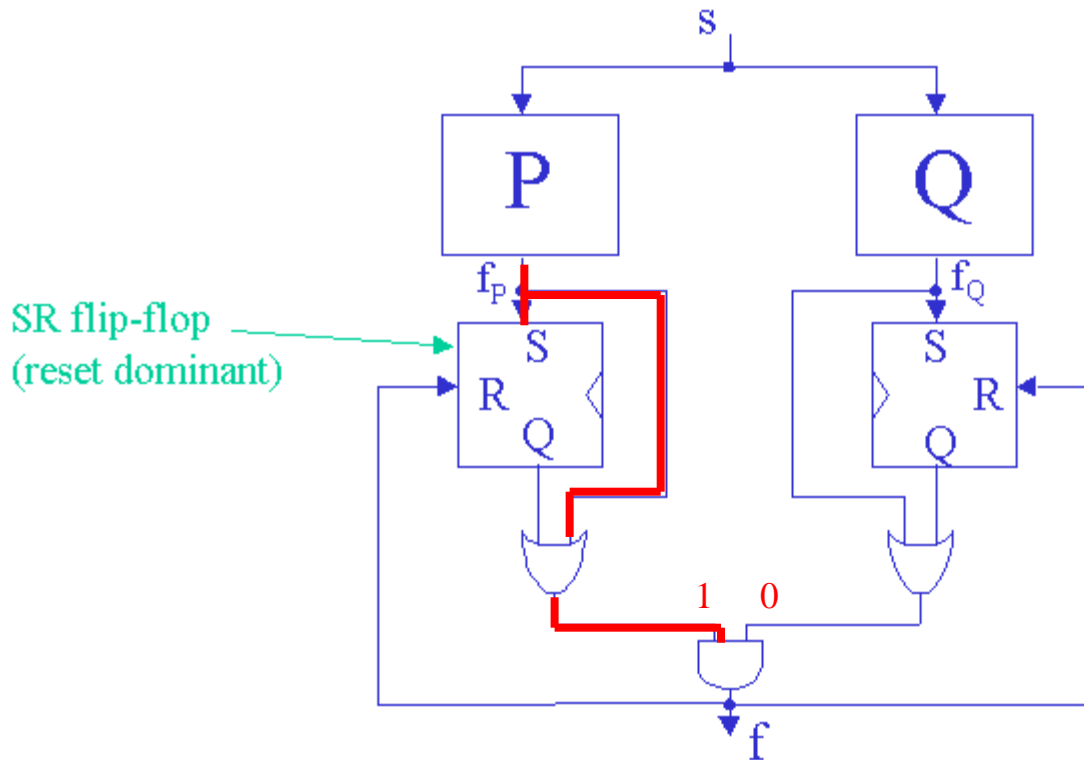


Summary of previous lecture

- hardware compilation
 - program directly to hardware; token-passing control
 - data processor: avoid instruction fetch/decode, specialise ALU
 - exploit parallelism, pipelining, locality
 - target latest FPGA: millions of programmable gates/connections
- compilation strategy for parallel imperative language
 - variables → registers, expressions → combinational circuit
 - control circuit: token to activate corresponding datapath
 - assignment: mux to select value, DFF to indicate completion
 - seq: compose control circuits
 - if/while: demux to route token, or-gate to merge token
 - parallel: start concurrently, SRFF records end, or-gate by-pass

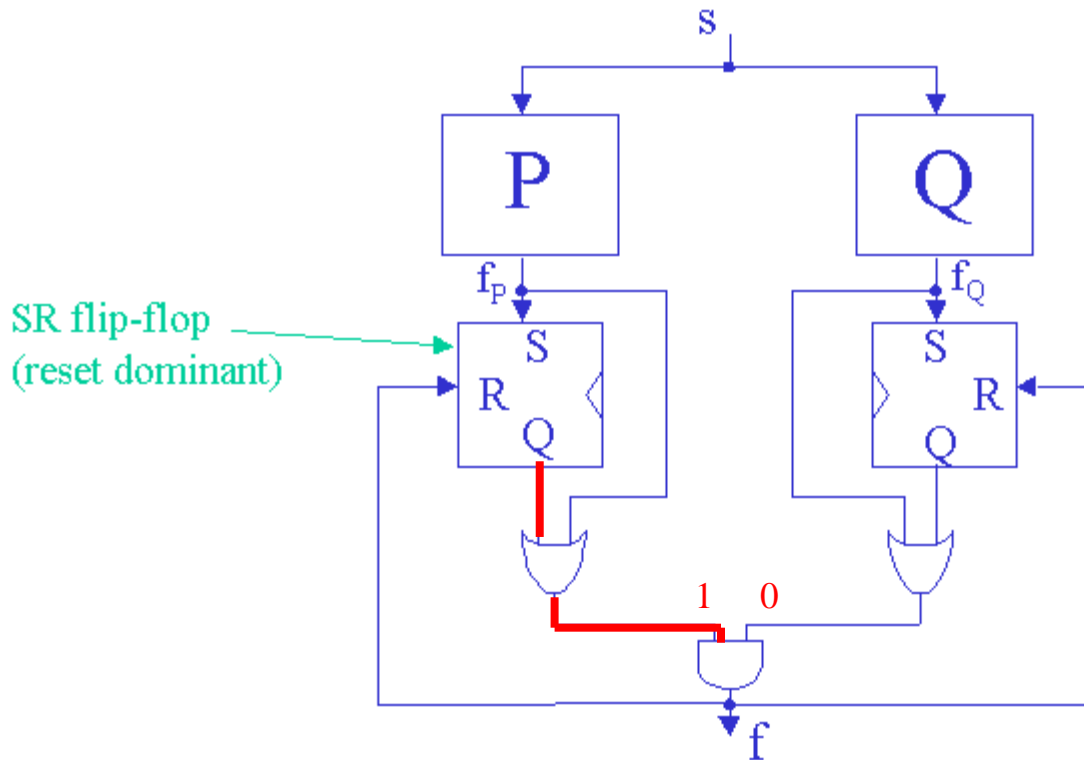
Control circuit for par, P finishes first



- PAR
 - P
 - Q
- or
- par { P;
 - Q

- SR flip-flop: once set, remains set until R input high
- or-gate: allows immediate termination

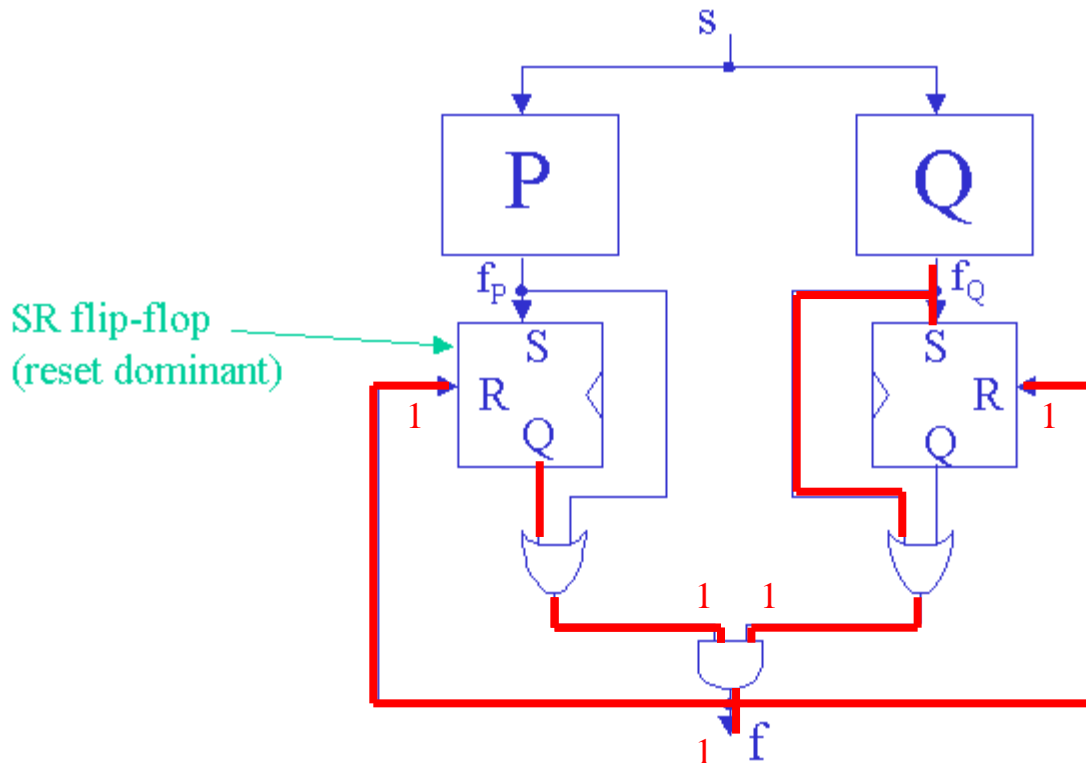
Control circuit for par, P waits for Q



- PAR
P
Q
- or
- par { P;
Q
}

- SR flip-flop: once set, remains set until R input high
- or-gate: allows immediate termination

Control circuit for par, Q finishes



- PAR
 - P
 - Q
- or
- par { P;
 - Q

- SR flip-flop: once set, remains set until R input high
- or-gate: allows immediate termination

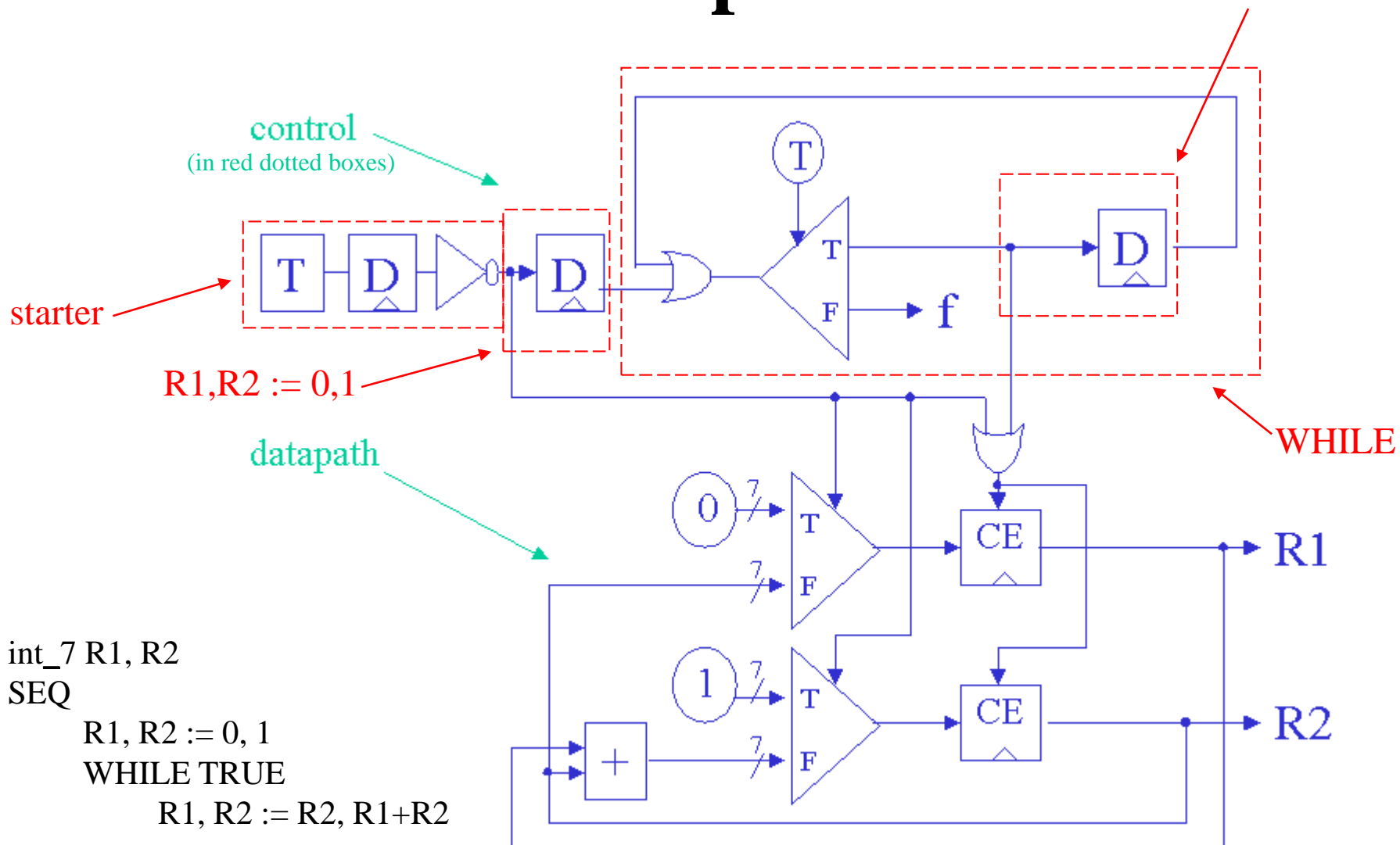
Example: Fibonacci program

- int_7 R1, R2 } variable declaration
SEQ (generate 7-bit registers)
R1, R2 := 0, 1 } init.
WHILE TRUE } compute } generate
R1, R2 := R2, R1+R2 } next fib. } control
hardware,
datapath

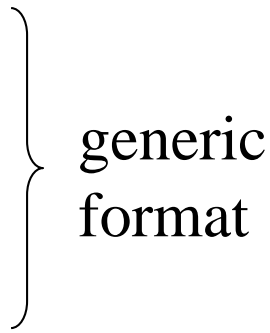
- | time= | 0 | R1 = | 0 | R2 = | 1 |
|-------|---|------|---|------|---|
| | 1 | | 1 | | 1 |
| | 2 | | 1 | | 2 |
| | 3 | | 2 | | 3 |
| | 4 | | 3 | | 5 |
| | 5 | | 5 | | 8 |

Fibonacci data processor

$R1, R2 := R2, R1 + R2$



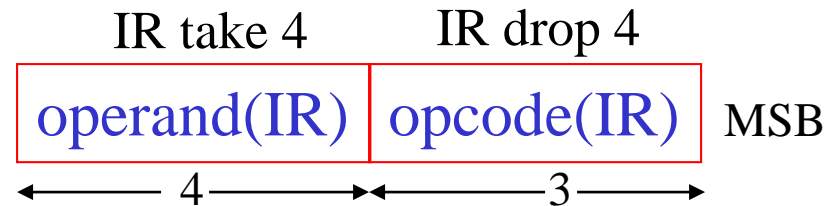
Compiling instruction processors

- define instruction set, possibly application dependent
- develop interpreter for instruction, e.g.
 - variable declaration (e.g. PC, IR)
 - WHILE TRUE
 - SEQ
 - fetch_next_instruction
 - decode_and_execute

generic format
- convert application programs into machine code;
compile instruction interpreter into hardware
- facilitate customisation of resources to application
 - remove resources for unused instructions
 - combine groups of instructions into a new instruction

Fibonacci instruction processor

- 7 instructions



4-bit register → 7-bit register →

- int_4 PC: int_7 IR, A: [16] int_7 M:

WHILE TRUE
SEQ

PC, IR := PC + 1, M[PC]

CASE (opcode (IR))

0: SKIP

1: A := operand(IR) -- load const

2: A := M[operand(IR)] -- load accum

3: M[operand(IR)] := A -- store accum

4: A := A + M[operand(IR)]

5: PC := operand(IR)

6: IF A < 0 THEN PC := operand(IR)

← fetch

← decode, exec.

← SKIP instr.

← LDC instr.

← LDA instr.

← STA instr.

← ADDA instr.

← JMP instr.

← JLT instr.

← array of 16 7-bit registers,
initialised with user program

Fibonacci machine code

address	M[address]	behaviour	high-level behaviour
0	LDC 0	$A := 0$	$R1 := 0$
1	STA R1	$R1 := A$	
2	LDC 1	$A := 1$	$R2 := 1$
3	STA R2	$R2 := A$	
4	ADDA R1	$A := A + R1$	$x := R1 + R2$
5	STA x	$x := A$	
6	LDA R2	$A := R2$	$R1 := R2$
7	STA R1	$R1 := A$	
8	LDA x	$A := x$	$R2 := x$ (with $R2 := A$ at address 3)
9	JMP 3	goto 3	
13	VAR R1	variables	loop contains 7 instructions, 2 cycles per instruction
14	VAR R2		
15	VAR x		


Variations of instruction processors

- example: repeatedly calculate $x^2 + y^2$
- possible implementations:
 1. add instruction, square by repeated add, accumulator or load-store
 2. add and square instructions, accumulator or load-store
 3. custom sumsq instruction: dedicated circuit for $x^2 + y^2$
 4. pipelined version of 3: overlap fetch/decode/execute
- current research
 - automatic identification of number/type of custom instructions, see FISH and CHIPS at course homepage
 - domain-specific optimisations for speed, size, energy/power... e.g. Custard for multi-threading, Arvand for machine learning
 - automatic generation of compile-time and run-time tools

Largest, fastest: 938 gates or Registers, 14 cycles

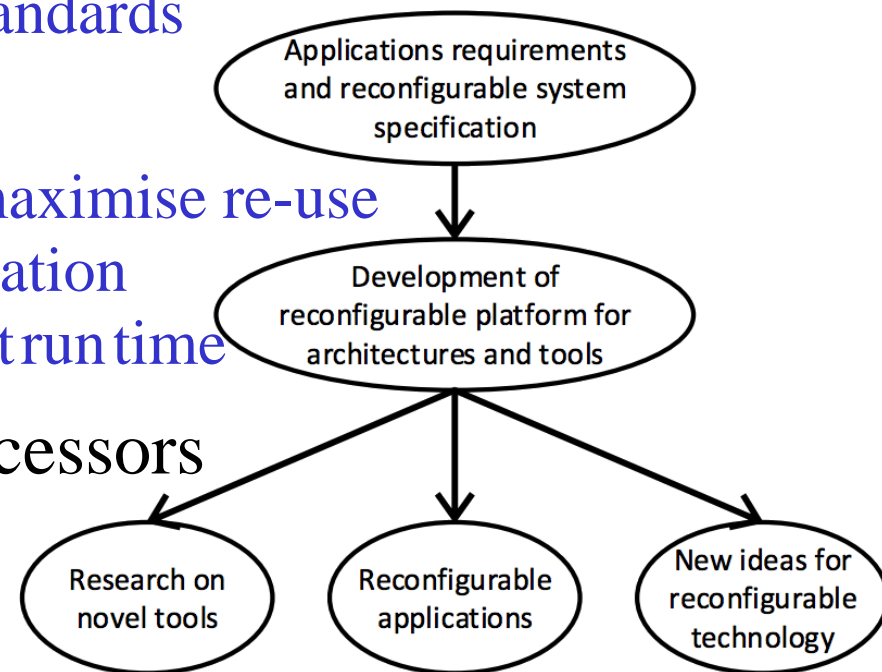


Smallest, slowest: 500 gates or registers, 14 cycles

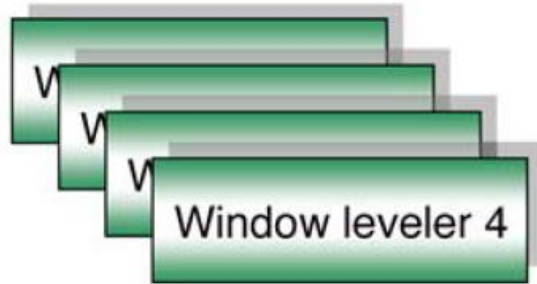


Reconfiguration: virtualise hardware

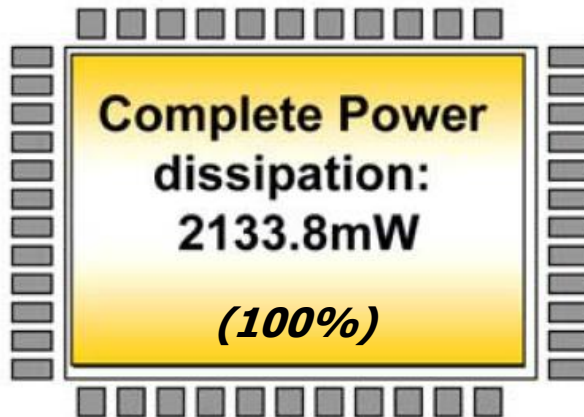
- exploit reconfigurability
 - not all operations are active all the time
 - download new operations while in service
 - reduce size, reduce power consumption, increase flexibility
- applications
 - adapt to environment, e.g. error correction based on noise level
 - partition large *virtual* hardware for small physical device
 - different (future) data formats/standards
- partially reconfigure systems
 - minimise reconfigurable parts, maximise re-use
 - overlap reconfiguration and operation
 - store configurations or synthesis at run time
- research on reconfigurable processors
 - <http://www.fp7-faster.eu>
 - <https://www.extrahpc.eu>



Benefits of reconfiguration: car control



All necessary modules resident on FPGA:



Only actually used modules configured on FPGA:

